```python
# -*- coding: utf-8 -*-
### SCRIPT 15 - PYTHON
# PIPELINE DESARROLLO DE MODELOS DE PREDICCION, CLUSTERS, kNN Y CENTROIDES
#
# ====================================================================

# ============================
# Celda A+C — PRE (TRAIN-only) + Centroides + POST FULL (TRAIN & ALL)
# ============================
# BASE: /content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2
# Entradas:
#   - <BASE>/inputs/df_test_train_v8.xlsx
# Salidas (PROD):
#   - <BASE>/modelos/preprocess_pipeline.pkl
#   - <BASE>/modelos/centroides_pre.json
# Salidas (VALID):
#   - <BASE>/validacion/01_PRE_CENTROIDES/outputs/centroides_pre_train_matrix.csv
#   - <BASE>/validacion/01_PRE_CENTROIDES/outputs/centroides_post_train_full.csv
#   - <BASE>/validacion/01_PRE_CENTROIDES/outputs/centroides_post_all_full.csv
#   - <BASE>/validacion/01_PRE_CENTROIDES/README_STEP.txt

import os, json, joblib
import numpy as np
import pandas as pd
from datetime import datetime

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

# Montaje Drive
try:
    from google.colab import drive  # type: ignore
    drive.mount('/content/drive')
except Exception:
    pass

# === Rutas base (TODO bajo _v2) ===
BASE_DIR   = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
ORIGEN_DIR = BASE_DIR
DIR_MODELOS = f"{BASE_DIR}/modelos"
DIR_VALID   = f"{BASE_DIR}/validacion"
DIR_COMP    = f"{BASE_DIR}/comparativa"  # opcional (no escribimos por defecto)

STEP_TAG = "01_PRE_CENTROIDES"
STEP_DIR = os.path.join(DIR_VALID, STEP_TAG)
OUT_DIR  = os.path.join(STEP_DIR, "outputs")
os.makedirs(DIR_MODELOS, exist_ok=True)
os.makedirs(OUT_DIR,      exist_ok=True)

random_state = 42

print(f"🏁 BASE_DIR: {BASE_DIR}")
print(f"📦 DIR_MODELOS: {DIR_MODELOS}")
print(f"🧪 DIR_VALID:   {DIR_VALID}")

# ==========
# 1) Datos
# ==========
df_path_candidates = [
    f"{ORIGEN_DIR}/inputs/df_test_train_v8.xlsx",
    f"{ORIGEN_DIR}/df_test_train_v8.xlsx",
]
df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
assert df_path is not None, f"No se encontró df_test_train_v8.xlsx en: {df_path_candidates}"
df = pd.read_excel(df_path)
print(f"✓ df_test_train_v8.xlsx cargado: {df.shape}")

# Checks mínimos
if 'Cluster_6' not in df.columns:
    raise ValueError("❌ Falta 'Cluster_6'.")
if 'Dataset' not in df.columns:
    raise ValueError("❌ Falta 'Dataset' (TRAIN/TEST).")

df['Cluster_6'] = df['Cluster_6'].astype(str)
df['Dataset']   = df['Dataset'].astype(str).str.upper()

# ==========
# 2) Variables PRE
# ==========
pre_num = [
    "N_lotes","C_precio_p","N_CPV","N_clasi_empresa","Presupuesto_licitacion_lote_c",
    "Plazo_m_c","C_juicio_valor_p_c","Intervalo_lici_d_c",
]
pre_cat = [
    "Tipo_de_contrato_c","Tipo_de_Administracion_c","Tipo_de_procedimiento_c",
    "Tramitacion_c",
    # Provincia2 se añade si falta
    "Tipo_ganador_lote_c","Mes_lici",
]

# Derivar Provincia2 (dos primeros dígitos)
def _to_cp_str(x):
    try:
        return str(int(x)).zfill(5)
    except Exception:
        s = str(x); return s if s and s.lower() != 'nan' else None

if "Provincia2" not in df.columns:
    if "Codigo_Postal_c" in df.columns:
        prov = df["Codigo_Postal_c"].apply(_to_cp_str).str[:2]
        df["Provincia2"] = prov.where(prov.notna(), np.nan)
    else:
        df["Provincia2"] = np.nan
if "Provincia2" not in pre_cat:
    pre_cat = pre_cat[:4] + ["Provincia2"] + pre_cat[4:]

# Completar columnas ausentes
for c in pre_num:
    if c not in df.columns: df[c] = np.nan
for c in pre_cat:
    if c not in df.columns: df[c] = np.nan

# ==========
# 3) PREPROCESSOR (TRAIN-only)
# ==========
df_train = df[df["Dataset"] == "TRAIN"].copy()
df_all   = df.copy()

num_pipe = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler",  StandardScaler())
])
cat_pipe = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("ohe",     OneHotEncoder(handle_unknown="ignore", sparse_output=False))
])

preprocessor = ColumnTransformer(
    transformers=[("num", num_pipe, pre_num), ("cat", cat_pipe, pre_cat)],
    remainder='drop',
    verbose_feature_names_out=True
)

X_train_pre = df_train[pre_num + pre_cat].copy()
y_train     = df_train["Cluster_6"].astype(str).copy()

preprocessor.fit(X_train_pre, y_train)
feature_names = preprocessor.get_feature_names_out().tolist()
print(f"• Features transformadas: {len(feature_names)}")

# ==========
# 4) Centroides PRE (espacio transformado, TRAIN-only)
# ==========
Z_train = preprocessor.transform(X_train_pre)
classes = sorted(y_train.unique(), key=lambda x: int(x))

centroids_pre, p95_intra = {}, {}
for c in classes:
    Zc = Z_train[(y_train == c).values]
    mu = Zc.mean(axis=0)
    centroids_pre[c] = mu.tolist()
    d = np.sqrt(((Zc - mu) ** 2).sum(axis=1))
    p95_intra[c] = float(np.percentile(d, 95))

# Guardar artefactos (PROD)
joblib.dump(preprocessor, f"{DIR_MODELOS}/preprocess_pipeline.pkl")
with open(f"{DIR_MODELOS}/centroides_pre.json", "w", encoding="utf-8") as f:
    json.dump({
        "classes": classes,
        "feature_names": feature_names,
        "centroids_pre": centroids_pre,
        "p95_intra": p95_intra,
        "random_state": random_state,
        "built_at": datetime.now().isoformat()
    }, f, ensure_ascii=False, indent=2)
print("💾 Guardado (PROD): preprocess_pipeline.pkl, centroides_pre.json")

# Copias de validación
cent_pre_mat = pd.DataFrame.from_dict(centroids_pre, orient='index', columns=feature_names)
cent_pre_mat.index.name = "Cluster_6"
cent_pre_mat.sort_index(key=lambda x: x.map(int), inplace=True)
cent_pre_mat.to_csv(f"{OUT_DIR}/centroides_pre_train_matrix.csv")

def _top12_share(series: pd.Series):
    vc = series.value_counts(dropna=True)
    if len(vc) == 0: return (np.nan, np.nan, np.nan, np.nan)
    top1, s1 = vc.index[0], float(vc.iloc[0]) / float(vc.sum())
    if len(vc) > 1:
        top2, s2 = vc.index[1], float(vc.iloc[1]) / float(vc.sum())
    else:
        top2, s2 = (np.nan, np.nan)
    return (top1, s1, top2, s2)

def _post_full(df_base: pd.DataFrame, idx_vals):
    pre_num_present = [c for c in pre_num if c in df_base.columns]
    post_num = [c for c in ["N_ofertantes","Ofertas_admitidas","Desierta", "Importe_adjudicacion_lote"] if c in df_base.columns]
    out = pd.DataFrame(index=idx_vals)
    if pre_num_present:
        g = df_base.groupby("Cluster_6")
        pre_mean = g[pre_num_present].mean(numeric_only=True).add_prefix("PRE__").add_suffix("__mean")
        pre_std  = g[pre_num_present].std(numeric_only=True).add_prefix("PRE__").add_suffix("__std")
        out = out.join(pre_mean, how="left").join(pre_std, how="left")
    if post_num:
        g = df_base.groupby("Cluster_6")
        post_mean = g[post_num].mean(numeric_only=True).add_prefix("POST__").add_suffix("__mean")
        post_std  = g[post_num].std(numeric_only=True).add_prefix("POST__").add_suffix("__std")
        out = out.join(post_mean, how="left").join(post_std,  how="left")
    rows = []
    for c in idx_vals:
        sub = df_base[df_base["Cluster_6"].astype(str) == c]
        row = {"Cluster_6": c}
        for col in pre_cat:
            if col not in sub.columns:
                row[f"PRECAT__{col}__top1"]=np.nan; row[f"PRECAT__{col}__top1_share"]=np.nan
                row[f"PRECAT__{col}__top2"]=np.nan; row[f"PRECAT__{col}__top2_share"]=np.nan
            else:
                top1, s1, top2, s2 = _top12_share(sub[col])
                row[f"PRECAT__{col}__top1"]=top1; row[f"PRECAT__{col}__top1_share"]=s1
                row[f"PRECAT__{col}__top2"]=top2; row[f"PRECAT__{col}__top2_share"]=s2
        rows.append(row)
    out2 = pd.DataFrame(rows).set_index("Cluster_6")
    final = out.join(out2, how="left")
    final.index.name = "Cluster_6"
    final.sort_index(key=lambda x: x.map(int), inplace=True)
    return final

post_train = _post_full(df_train.copy(), classes)
post_all   = _post_full(df_all.copy(),   classes)
post_train.to_csv(f"{OUT_DIR}/centroides_post_train_full.csv")
post_all.to_csv(f"{OUT_DIR}/centroides_post_all_full.csv")

with open(os.path.join(STEP_DIR, "README_STEP.txt"), "w", encoding="utf-8") as f:
    f.write(f"""STEP: {STEP_TAG}
BASE_DIR: {BASE_DIR}
Fecha: {datetime.now().isoformat()}
random_state: {random_state}
Artefactos PROD: modelos/preprocess_pipeline.pkl, modelos/centroides_pre.json
Salidas VALID: outputs/*.csv
""")
print("✅ Celda A+C COMPLETADA (todo bajo _v2).")

"""Celda D — Distancias a centroides (TRAIN/TEST)"""

# ============================
# Celda D — Distancias a centroides PRE (TRAIN/TEST)
# ============================
# Entradas:
#   - <BASE>/inputs/df_test_train_v8.xlsx
#   - <BASE>/modelos/preprocess_pipeline.pkl
#   - <BASE>/modelos/centroides_pre.json
# Salidas (VALID):
#   - <BASE>/validacion/03_DISTANCIAS_PRE/outputs/df_train_with_dist.csv
#   - <BASE>/validacion/03_DISTANCIAS_PRE/outputs/df_test_with_dist.csv
#   - <BASE>/validacion/03_DISTANCIAS_PRE/outputs/summary_distancias.json

import os, json, joblib
import numpy as np
import pandas as pd
from datetime import datetime

try:
    from google.colab import drive  # type: ignore
    drive.mount('/content/drive')
except Exception:
    pass

BASE_DIR   = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
ORIGEN_DIR = BASE_DIR
DIR_MODELOS = f"{BASE_DIR}/modelos"
DIR_VALID   = f"{BASE_DIR}/validacion"

STEP_TAG = "03_DISTANCIAS_PRE"
STEP_DIR = os.path.join(DIR_VALID, STEP_TAG)
OUT_DIR  = os.path.join(STEP_DIR, "outputs")
os.makedirs(OUT_DIR, exist_ok=True)

# Artefactos
preproc_path   = os.path.join(DIR_MODELOS, "preprocess_pipeline.pkl")
cent_json_path = os.path.join(DIR_MODELOS, "centroides_pre.json")
assert os.path.exists(preproc_path), "Falta preprocess_pipeline.pkl"
assert os.path.exists(cent_json_path), "Falta centroides_pre.json"

preprocessor = joblib.load(preproc_path)
with open(cent_json_path, "r", encoding="utf-8") as f:
    cent_meta = json.load(f)

classes       = [str(c) for c in cent_meta["classes"]]
```

```
centroids_pre = {str(k): np.array(v, dtype=float) for k, v in cent_meta[
"centroids_pre"].items()}

# Datos
df_path_candidates = [
    f"{ORIGEN_DIR}/inputs/df_test_train_v8.xlsx",
    f"{ORIGEN_DIR}/df_test_train_v8.xlsx",
]
df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
assert df_path is not None, "No se encontró df_test_train_v8.xlsx"
df = pd.read_excel(df_path)
if "Cluster_6" in df.columns: df["Cluster_6"] = df["Cluster_6"].astype(str)
df["Dataset"] = df["Dataset"].astype(str).str.upper()

pre_num = [
    "N_lotes","C_precio_p","N_CPV","N_clasi_empresa","Presupuesto_licitacion_lote_c",
    "Plazo_m_c","C_juicio_valor_p_c","Intervalo_lici_d_c",
]
pre_cat = [
    "Tipo_de_contrato_c","Tipo_de_Administracion_c","Tipo_de_procedimiento_c",
    "Tramitacion_c",
    "Provincia2","Tipo_ganador_lote_c","Mes_lici",
]

def _to_cp_str(x):
    try:
        return str(int(x)).zfill(5)
    except Exception:
        s = str(x); return s if s and s.lower() != 'nan' else None

def _prepare_pre(df_in: pd.DataFrame) -> pd.DataFrame:
    df = df_in.copy()
    if "Provincia2" not in df.columns:
        if "Codigo_Postal_c" in df.columns:
            prov = df["Codigo_Postal_c"].apply(_to_cp_str).str[:2]
            df["Provincia2"] = prov.where(prov.notna(), np.nan)
        else:
            df["Provincia2"] = np.nan
    if "Mes_lici" not in df.columns: df["Mes_lici"] = np.nan
    for c in pre_num:
        if c not in df.columns: df[c] = np.nan
    for c in pre_cat:
        if c not in df.columns: df[c] = np.nan
    return df[pre_num + pre_cat]

df_train = df[df["Dataset"] == "TRAIN"].copy()
df_test  = df[df["Dataset"] == "TEST"].copy()
Z_train  = preprocessor.transform(_prepare_pre(df_train))
Z_test   = preprocessor.transform(_prepare_pre(df_test))

def _dist_to_centroids(Z):
    rows=[]
    for i in range(Z.shape[0]):
        z=Z[i]; dists={}
        for c in classes:
            mu = centroids_pre[c]
            dists[f"dist_c{c}"] = float(np.sqrt(((z - mu) ** 2).sum()))
        items = sorted(dists.items(), key=lambda x: x[1])
        min_k, min_d = items[0]; second_d = items[1][1] if len(items)>1 else np.nan
        rows.append({"min_cluster": min_k.replace("dist_c",""), "min_dist": min_d,
        "margin": float(second_d-min_d) if np.isfinite(second_d) else np.nan, **dists
        })
    return pd.DataFrame(rows)

dist_train = _dist_to_centroids(Z_train)
dist_test  = _dist_to_centroids(Z_test)

train_out = pd.concat([df_train.reset_index(drop=True), dist_train], axis=1)
test_out  = pd.concat([df_test.reset_index(drop=True),  dist_test],  axis=1)

train_out_path = os.path.join(OUT_DIR, "df_train_with_dist.csv")
test_out_path  = os.path.join(OUT_DIR, "df_test_with_dist.csv")
```

```python
train_out.to_csv(train_out_path, index=False)
test_out.to_csv(test_out_path,  index=False)

with open(os.path.join(OUT_DIR, "summary_distancias.json"), "w", encoding="utf-8") as f:
    json.dump({
        "step": STEP_TAG,
        "built_at": datetime.now().isoformat(),
        "outputs": {"train_with_dist": train_out_path, "test_with_dist": test_out_path}
    }, f, ensure_ascii=False, indent=2)

with open(os.path.join(STEP_DIR, "README_STEP.txt"), "w", encoding="utf-8") as f:
    f.write(f"{STEP_TAG} generado el {datetime.now().isoformat()} (BASE_DIR={BASE_DIR}).\n")

print("✅ Distancias PRE generadas (todo bajo _v2).")

"""Celda A — META del PRE + centroides (preprocess_pipeline)"""

# ============================================
# META — PREPROCESS + CENTROIDES (guardado)
# ============================================
import json, joblib, numpy as np
from pathlib import Path

BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE / "modelos"
DIR_MODELOS.mkdir(parents=True, exist_ok=True)

preproc_path   = DIR_MODELOS / "preprocess_pipeline.pkl"
centroids_path = DIR_MODELOS / "centroides_pre.json"
assert preproc_path.exists() and centroids_path.exists(), "Faltan preprocess_pipeline.pkl y/o centroides_pre.json"

pre = joblib.load(preproc_path)
cent = json.loads(centroids_path.read_text(encoding="utf-8"))

# Extraer columnas crudas esperadas por el ColumnTransformer
num_cols = []
cat_cols = []
try:
    trfs = pre.named_transformers_
    if "num" in trfs: num_cols = list(pre.transformers_[0][2]) if isinstance(pre.transformers_[0][2], list) else []
    if "cat" in trfs: cat_cols = list(pre.transformers_[1][2]) if isinstance(pre.transformers_[1][2], list) else []
except Exception:
    pass

pre_meta = {
    "artifact": "preprocess_pipeline.pkl",
    "expects_raw_features": {
        "num": num_cols,
        "cat": cat_cols
    },
    "transformers": {
        "num": [s[0] for s in getattr(pre.named_transformers_.get("num", None), "steps", [])] if "num" in pre.named_transformers_ else [],
        "cat": [s[0] for s in getattr(pre.named_transformers_.get("cat", None), "steps", [])] if "cat" in pre.named_transformers_ else []
    },
    "centroids_ref": "centroides_pre.json",
    "centroids_overview": {
        "classes": cent.get("classes"),
        "p95_intra_available": bool(cent.get("p95_intra"))
    }
}
(DIR_MODELOS / "preprocess_meta.json").write_text(json.dumps(pre_meta, indent=2, ensure_ascii=False), encoding="utf-8")
print("✅ Guardado modelos/preprocess_meta.json")
```

```python
"""Celda E — kNN PRE (TRAIN y ALL) — NUM-ONLY"""

# ============================
# Celda E — kNN de similares (TRAIN y ALL) — NUM-ONLY
# ============================
# Entradas:
#   - <BASE>/inputs/df_test_train_v8.xlsx
# Salidas (PROD):
#   - <BASE>/modelos/knn_pre_train.joblib
#   - <BASE>/modelos/knn_pre_all.joblib
#   - <BASE>/modelos/knn_params.json
# Salidas (VALID):
#   - <BASE>/validacion/04_KNN_PRE/outputs/knn_train_cases.parquet
#   - <BASE>/validacion/04_KNN_PRE/outputs/knn_all_cases.parquet
#   - <BASE>/validacion/04_KNN_PRE/outputs/meta_knn.json

import os, json, joblib
import numpy as np
import pandas as pd
from datetime import datetime
from sklearn.neighbors import NearestNeighbors
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

try:
    from google.colab import drive  # type: ignore
    drive.mount('/content/drive')
except Exception:
    pass

BASE_DIR   = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
ORIGEN_DIR = BASE_DIR
DIR_MODELOS = f"{BASE_DIR}/modelos"
DIR_VALID   = f"{BASE_DIR}/validacion"

STEP_TAG = "04_KNN_PRE"
STEP_DIR = os.path.join(DIR_VALID, STEP_TAG)
OUT_DIR  = os.path.join(STEP_DIR, "outputs")
os.makedirs(DIR_MODELOS, exist_ok=True)
os.makedirs(OUT_DIR,      exist_ok=True)

# Datos
df_path_candidates = [
    f"{ORIGEN_DIR}/inputs/df_test_train_v8.xlsx",
    f"{ORIGEN_DIR}/df_test_train_v8.xlsx",
]
df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
assert df_path is not None, "No se encontró df_test_train_v8.xlsx"
df = pd.read_excel(df_path)
df["Dataset"] = df["Dataset"].astype(str).str.upper()

df_train = df[df["Dataset"] == "TRAIN"].copy()
df_test  = df[df["Dataset"] == "TEST"].copy()
df_all   = pd.concat([df_train, df_test], axis=0, ignore_index=True)

# PRE NUM-ONLY
pre_num = [
    "N_lotes","C_precio_p","N_CPV","N_clasi_empresa","Presupuesto_licitacion_lote_c",
    "Plazo_m_c","C_juicio_valor_p_c","Intervalo_lici_d_c",
]

def _prepare_num(df_in: pd.DataFrame) -> pd.DataFrame:
    df = df_in.copy()
    for c in pre_num:
        if c not in df.columns: df[c] = np.nan
        df[c] = pd.to_numeric(df[c], errors="coerce")
    return df[pre_num]

X_train_num = _prepare_num(df_train)
X_all_num   = _prepare_num(df_all)
```

```python
num_only = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler",  StandardScaler())
])
num_only.fit(X_train_num)  # TRAIN-only

Z_train = num_only.transform(X_train_num)
Z_all   = num_only.transform(X_all_num)

# kNN + p95
K_NEIGH = 25
nn_train = NearestNeighbors(n_neighbors=K_NEIGH, metric="euclidean").fit(Z_train)
nn_all   = NearestNeighbors(n_neighbors=K_NEIGH, metric="euclidean").fit(Z_all)

nn_tmp   = NearestNeighbors(n_neighbors=2, metric="euclidean").fit(Z_train)
dist2, _ = nn_tmp.kneighbors(Z_train, n_neighbors=2, return_distance=True)
nn1_dists = dist2[:, 1]
p95_nn1_train = float(np.percentile(nn1_dists, 95))
print(f"🔎 p95_nn1_train (NUM-ONLY) = {p95_nn1_train:.4f}")

# Guardar artefactos PROD
joblib.dump(nn_train, os.path.join(DIR_MODELOS, "knn_pre_train.joblib"))
joblib.dump(nn_all,   os.path.join(DIR_MODELOS, "knn_pre_all.joblib"))
with open(os.path.join(DIR_MODELOS, "knn_params.json"), "w", encoding="utf-8") as f:
    json.dump({
        "built_at": datetime.now().isoformat(),
        "k_neighbors": K_NEIGH,
        "p95_nn1_train": p95_nn1_train,
        "metric": "euclidean",
        "index_key": "knn_row",
        "space": "pre_num",
        "pre_num_features": pre_num
    }, f, ensure_ascii=False, indent=2)
print("💾 Guardado (PROD): knn_pre_train.joblib, knn_pre_all.joblib, knn_params.json")

# Casos VALID
def _attach_knn_row(df_src: pd.DataFrame) -> pd.DataFrame:
    out = df_src.reset_index(drop=True).copy()
    out.insert(0, "knn_row", np.arange(len(out), dtype=int))
    return out

cases_train = _attach_knn_row(df_train)
cases_all   = _attach_knn_row(df_all)

train_cases_path = os.path.join(OUT_DIR, "knn_train_cases.parquet")
all_cases_path   = os.path.join(OUT_DIR, "knn_all_cases.parquet")
cases_train.to_parquet(train_cases_path, index=False)
cases_all.to_parquet(all_cases_path, index=False)

with open(os.path.join(OUT_DIR, "meta_knn.json"), "w", encoding="utf-8") as f:
    json.dump({
        "artifacts": {"nn_train": "modelos/knn_pre_train.joblib", "nn_all": "modelos/knn_pre_all.joblib"},
        "cases": {"train": train_cases_path, "all": all_cases_path, "format": "parquet", "index_key": "knn_row"},
        "p95_nn1_train": p95_nn1_train, "space": "pre_num"
    }, f, ensure_ascii=False, indent=2)

with open(os.path.join(STEP_DIR, "README_STEP.txt"), "w", encoding="utf-8") as f:
    f.write(f"{STEP_TAG} (NUM-ONLY) generado el {datetime.now().isoformat()} (BASE_DIR={BASE_DIR}).\n")

print("✅ kNN NUM-ONLY listo (todo bajo _v2).")

"""Celda B — META del kNN (incluye pipeline num-only fitted)"""

# ===========================================
# META — KNN (pipeline num-only + meta JSON)
# ===========================================
import json, joblib
from pathlib import Path

```

```python
BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE / "modelos"
assert 'num_only' in globals(), "num_only no está en memoria: ejecuta esta celda justo tras la Celda E (KNN)."

# Guardar pipeline num-only usada realmente
knn_num_pipe_path = DIR_MODELOS / "knn_num_pipeline.joblib"
joblib.dump(num_only, knn_num_pipe_path)

# Actualizar/crear meta del KNN (aprovecha knn_params.json creado en Celda E)
knn_params_path = DIR_MODELOS / "knn_params.json"
assert knn_params_path.exists(), "Falta modelos/knn_params.json (Célula E)."
params = json.loads(knn_params_path.read_text(encoding="utf-8"))
params["num_only_pipeline_file"] = "knn_num_pipeline.joblib"
params["note"] = "Pipeline num-only exacta usada para estandarizar las 8 features (NUM-ONLY) antes del índice."
knn_params_path.write_text(json.dumps(params, indent=2, ensure_ascii=False), encoding="utf-8")

print("✅ Guardado modelos/knn_num_pipeline.joblib y actualizado modelos/knn_params.json")

"""Celda F — Validación kNN (ALL vs TRAIN) con TEST"""

# =============================================
# Celda E+1 — Catálogo kNN (ALL) COMPLETO
# =============================================
# BASE: /content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2
# Entradas:
#   - <BASE>/inputs/df_test_train_v8.xlsx
#   - (opcional) <BASE>/validacion/04_KNN_PRE/outputs/knn_all_cases.parquet  (para verificar alineación)
# Salidas (PROD):
#   - <BASE>/modelos/knn_catalog_all.parquet        (TODAS las variables + knn_row)
#   - <BASE>/modelos/knn_catalog_meta.json
# Salidas (VALID):
#   - <BASE>/validacion/04_KNN_PRE/outputs/knn_catalog_all_head.csv  (muestra)
#   - <BASE>/validacion/04_KNN_PRE/outputs/knn_catalog_alignment.json (reporte verificación)

import os, json
import numpy as np
import pandas as pd
from datetime import datetime

# Rutas base
BASE_DIR    = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
DIR_MODELOS = f"{BASE_DIR}/modelos"
DIR_VALID   = f"{BASE_DIR}/validacion"
STEP_DIR    = os.path.join(DIR_VALID, "04_KNN_PRE")
OUT_DIR     = os.path.join(STEP_DIR, "outputs")
os.makedirs(DIR_MODELOS, exist_ok=True)
os.makedirs(OUT_DIR, exist_ok=True)

# 1) Cargar df y recrear ALL (TRAIN+TEST)
df_path_candidates = [
    f"{BASE_DIR}/inputs/df_test_train_v8.xlsx",
    f"{BASE_DIR}/df_test_train_v8.xlsx",
]
df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
assert df_path is not None, "No se encontró df_test_train_v8.xlsx"
df = pd.read_excel(df_path)
assert "Dataset" in df.columns, "❌ Falta 'Dataset' (TRAIN/TEST)"
df["Dataset"] = df["Dataset"].astype(str).str.upper()

df_train = df[df["Dataset"] == "TRAIN"].copy()
df_test  = df[df["Dataset"] == "TEST"].copy()
df_all   = pd.concat([df_train, df_test], axis=0, ignore_index=True)

# 2) Asegurar Provincia2 si falta (regla del proyecto)
def _to_cp_str(x):
    try: return str(int(x)).zfill(5)
```

```python
        except Exception:
            s = str(x); return s if (isinstance(s, str) and s and s.lower()!="nan") else np.nan

    if "Provincia2" not in df_all.columns:
        if "Codigo_Postal_c" in df_all.columns:
            df_all["Provincia2"] = df_all["Codigo_Postal_c"].apply(_to_cp_str).str[:2]
        else:
            df_all["Provincia2"] = np.nan

    # 3) Asegurar columna de ID amigable (no sobrescribimos si ya existe)
    id_candidates = ["Identificador","identificador","ID","Id","id_lote","id_contrato"]
    id_col = next((c for c in id_candidates if c in df_all.columns), None)
    if id_col is None:
        df_all["Identificador"] = [f"row_{i}" for i in range(len(df_all))]
        id_col = "Identificador"

    # 4) Insertar knn_row para alinear con Celda E (reset_index + rango continuo)
    catalog = df_all.reset_index(drop=True).copy()
    catalog.insert(0, "knn_row", np.arange(len(catalog), dtype=int))

    # 5) Guardar catálogo completo (PRODUCCIÓN)
    catalog_path = os.path.join(DIR_MODELOS, "knn_catalog_all.parquet")
    catalog.to_parquet(catalog_path, index=False)

    # 6) Muestra y metadatos (VALIDACIÓN)
    head_path = os.path.join(OUT_DIR, "knn_catalog_all_head.csv")
    catalog.head(30).to_csv(head_path, index=False)

    meta = {
        "built_at": datetime.now().isoformat(),
        "rows": int(len(catalog)),
        "cols": int(catalog.shape[1]),
        "id_col": id_col,
        "index_key": "knn_row",
        "source_df": os.path.basename(df_path),
        "note": "Catálogo ALL (TRAIN+TEST) con TODAS las variables + knn_row al frente."
    }
    with open(os.path.join(DIR_MODELOS, "knn_catalog_meta.json"), "w", encoding="utf-8") as f:
        json.dump(meta, f, ensure_ascii=False, indent=2)

    # 7) Verificación de alineación con knn_all_cases.parquet (si existe)
    align_report = {"checked": False, "ok": None, "details": {}}
    cases_all_path = os.path.join(OUT_DIR, "knn_all_cases.parquet")
    if os.path.exists(cases_all_path):
        cases_all = pd.read_parquet(cases_all_path)
        align_report["checked"] = True
        try:
            # mismas filas y mismo rango de knn_row
            same_len = (len(cases_all) == len(catalog))
            same_range = (cases_all["knn_row"].iloc[0] == 0 and cases_all["knn_row"].iloc[-1] == len(catalog)-1)
            # chequeo de identidad en 5 posiciones aleatorias
            rng = np.random.RandomState(42)
            sample_idx = rng.choice(len(catalog), size=min(5, len(catalog)), replace=False)
            id_match = []
            for i in sample_idx:
                r_cases = cases_all.iloc[i]
                r_cata  = catalog.iloc[i]
                id_match.append(bool(r_cases.get("knn_row", -1) == r_cata.get("knn_row", -2)))
            align_ok = same_len and same_range and all(id_match)
            align_report["ok"] = bool(align_ok)
            align_report["details"] = {
                "same_len": bool(same_len),
                "same_range": bool(same_range),
                "sample_id_match_rate": float(np.mean(id_match)) if len(id_match)>0 else None
            }
        except Exception as e:
```

```
        align_report["ok"] = False
        align_report["details"] = {"error": str(e)}

with open(os.path.join(OUT_DIR, "knn_catalog_alignment.json"), "w", encoding="utf-8") as f:
    json.dump(align_report, f, ensure_ascii=False, indent=2)

print("✅ Catálogo kNN ALL generado.")
print(f"→ modelos/knn_catalog_all.parquet  | filas={len(catalog)} cols={catalog.shape[1]}")
print(f"→ validacion/04_KNN_PRE/outputs/knn_catalog_all_head.csv")
if align_report["checked"]:
    print(f"✔ Verificación con knn_all_cases.parquet | ok={align_report['ok']} | details={align_report['details']}")
else:
    print("ℹ️ No se encontró knn_all_cases.parquet; se omitió verificación de alineación.")

"""Celda E+1 — Catálogo kNN (ALL) COMPLETO"""

# ============================
# Celda F — Validación kNN (ALL vs TRAIN) con TEST
# ============================
# Salidas (VALID):
#   - <BASE>/validacion/05_KNN_VALIDACION/outputs/knn_test_validation_pairs_all.csv
#   - <BASE>/validacion/05_KNN_VALIDACION/outputs/knn_test_validation_pairs_train.csv
#   - <BASE>/validacion/05_KNN_VALIDACION/outputs/knn_test_hit3_by_cluster_cases.csv
#   - <BASE>/validacion/05_KNN_VALIDACION/outputs/knn_validation_metrics_extended.json

import os, json, joblib
import numpy as np
import pandas as pd
from datetime import datetime
from sklearn.neighbors import NearestNeighbors

try:
    from google.colab import drive  # type: ignore
    drive.mount('/content/drive')
except Exception:
    pass

BASE_DIR   = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
ORIGEN_DIR = BASE_DIR
DIR_MODELOS = f"{BASE_DIR}/modelos"
DIR_VALID   = f"{BASE_DIR}/validacion"

STEP_TAG = "05_KNN_VALIDACION"
STEP_DIR = os.path.join(DIR_VALID, STEP_TAG)
OUT_DIR  = os.path.join(STEP_DIR, "outputs")
os.makedirs(OUT_DIR, exist_ok=True)

# Artefactos kNN
nn_train_path   = os.path.join(DIR_MODELOS, "knn_pre_train.joblib")
nn_all_path     = os.path.join(DIR_MODELOS, "knn_pre_all.joblib")
knn_params_path = os.path.join(DIR_MODELOS, "knn_params.json")
assert os.path.exists(nn_train_path) and os.path.exists(nn_all_path) and os.path.exists(knn_params_path), "Faltan artefactos kNN"

with open(knn_params_path, "r", encoding="utf-8") as f:
    knn_params = json.load(f)
p95_nn1_train = float(knn_params["p95_nn1_train"])
INDEX_KEY = knn_params.get("index_key", "knn_row")

nn_train = joblib.load(nn_train_path)
nn_all   = joblib.load(nn_all_path)

# Casos (de Celda E)
cases_train_path = os.path.join(DIR_VALID, "04_KNN_PRE", "outputs", "knn_train_cases.parquet")
cases_all_path   = os.path.join(DIR_VALID, "04_KNN_PRE", "outputs", "knn_all_cases.parquet")
cases_train = pd.read_parquet(cases_train_path)
```

```python
cases_all   = pd.read_parquet(cases_all_path)

# Datos TEST
df_path_candidates = [
    f"{ORIGEN_DIR}/inputs/df_test_train_v8.xlsx",
    f"{ORIGEN_DIR}/df_test_train_v8.xlsx",
]
df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
assert df_path is not None, "No se encontró df_test_train_v8.xlsx"
df = pd.read_excel(df_path)
df["Dataset"] = df["Dataset"].astype(str).str.upper()
df_test = df[df["Dataset"] == "TEST"].copy().reset_index(drop=True)

# Columna ID para excluir 'self' por ID
id_col = None
for cand in ["Identificador","identificador","ID","Id","id_lote","id_contrato"]:
    if cand in df_test.columns: id_col = cand; break
if id_col is None:
    df_test["Identificador"] = [f"test_row_{i}" for i in range(len(df_test))]
    id_col = "Identificador"

def sim_percent(d, p95=p95_nn1_train):
    return max(0.0, 1.0 - (float(d) / p95)) * 100.0 if p95 > 0 else 0.0

# Consultas (los objetos nn_* ya están ajustados sobre TRAIN/ALL en NUM-ONLY)
dist_all,  idx_all   = nn_all.kneighbors(n_neighbors=10,  return_distance=True, X=None)
dist_train,idx_train= nn_train.kneighbors(n_neighbors=10, return_distance=True, X=None)

# Construcción de pares
def build_pairs_csv(df_test, idx_mat, dist_mat, cases_df, out_csv_path, exclude_self_idx=None):
    rows=[]
    for i in range(len(df_test)):
        input_id = df_test.loc[i, id_col]
        pairs = [(float(dist_mat[i, j]), int(idx_mat[i, j])) for j in range(dist_mat.shape[1])]
        if exclude_self_idx is not None:
            pairs = [p for p in pairs if p[1] != exclude_self_idx(i)]
        if len(pairs) < 2: continue
        (d1, r1), (d2, r2) = pairs[0], pairs[1]
        row1 = cases_df.iloc[r1]; row2 = cases_df.iloc[r2]
        top1_id = row1.get(id_col, row1.get("Identificador", f"row_{r1}"))
        top2_id = row2.get(id_col, row2.get("Identificador", f"row_{r2}"))
        rows.append({"input_id": input_id,
                     "top1_id": top1_id, "top1_dist": d1, "top1_sim%": sim_percent(d1),
                     "top2_id": top2_id, "top2_dist": d2, "top2_sim%": sim_percent(d2)})
    out = pd.DataFrame(rows)
    out.to_csv(out_csv_path, index=False)
    return out

pairs_all_path = os.path.join(OUT_DIR, "knn_test_validation_pairs_all.csv")
pairs_all = build_pairs_csv(df_test, idx_all, dist_all, cases_all, pairs_all_path)
print(f"💾 {pairs_all_path}")

id_to_train_idx={}
if id_col in cases_train.columns:
    for i, r in cases_train.reset_index(drop=True).iterrows():
        id_to_train_idx[r.get(id_col, f"row_{i}")] = int(r[INDEX_KEY])

def exclude_self_idx(i):
    _id = df_test.loc[i, id_col]
    return id_to_train_idx.get(_id, -99999)

pairs_train_path = os.path.join(OUT_DIR, "knn_test_validation_pairs_train.csv")
pairs_train = build_pairs_csv(df_test, idx_train, dist_train, cases_train, pairs_train_path, exclude_self_idx=exclude_self_idx)
print(f"💾 {pairs_train_path}")

# Métricas rápidas
```

```python
SIM_THRESHOLD = 60.0
def hit_at_least_one(df_pairs, sim_thr=SIM_THRESHOLD):
    if df_pairs.empty: return 0.0
    return float((df_pairs[["top1_sim%","top2_sim%"]].max(axis=1) >= sim_thr).mean())*100.0

metrics = {
    "built_at": datetime.now().isoformat(),
    "p95_nn1_train": p95_nn1_train,
    "hit@2>=60_ALL(%)": hit_at_least_one(pairs_all),
    "hit@2>=60_TRAIN(%)": hit_at_least_one(pairs_train),
    "space": "pre_num"
}
with open(os.path.join(OUT_DIR, "knn_validation_metrics_extended.json"), "w", encoding="utf-8") as f:
    json.dump(metrics, f, ensure_ascii=False, indent=2)

# (Opcional) por cluster si existe
if "Cluster_6" in df_test.columns and len(pairs_all):
    tmp = pairs_all.copy()
    tmp["Cluster_6"] = df_test["Cluster_6"].values[:len(tmp)]
    agg = tmp.groupby("Cluster_6").apply(lambda g: float((g[["top1_sim%","top2_sim%"]].max(axis=1) >= SIM_THRESHOLD).mean())*100.0).reset_index()
    agg.columns = ["Cluster_6", "hit@2>=60(%)_ALL"]
    agg.to_csv(os.path.join(OUT_DIR, "knn_test_hit3_by_cluster_cases.csv"), index=False)

with open(os.path.join(STEP_DIR, "README_STEP.txt"), "w", encoding="utf-8") as f:
    f.write(f"{STEP_TAG} generado el {datetime.now().isoformat()} (BASE_DIR={BASE_DIR}).\n")

print("✅ Validación kNN completada (todo bajo _v2).")

"""Celda — XGB N_ofertantes (desarrollo + validación + guardados producción) desde df_test_train_v8.xlsx"""

# ================================================================
# XGB N_ofertantes — desarrollo + validación + guardados producción
# (desde df_test_train_v8.xlsx, sin usar comparativa como fuente)
# ================================================================
import os, json, joblib
import numpy as np
import pandas as pd
from pathlib import Path
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, f1_score, recall_score, confusion_matrix
import xgboost as xgb
import matplotlib.pyplot as plt

# --- Rutas base (todo bajo _v2) ---
BASE_DIR    = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE_DIR/"modelos"
DIR_VALID   = BASE_DIR/"validacion"
DIR_COMP    = BASE_DIR/"comparativa"   # solo para la celda de comparativa (no leemos aquí)
DIR_MODELOS.mkdir(parents=True, exist_ok=True)

# Validación: subcarpeta (mantengo el nombre que usaste en tu ejemplo)
vdir = DIR_VALID / "03_xgb_n_ofertantes"
vdir.mkdir(parents=True, exist_ok=True)

RANDOM_STATE = 42

# -------- 0) Carga df desde inputs --------
df_candidates = [BASE_DIR/"inputs/df_test_train_v8.xlsx", BASE_DIR/"df_test_train_v8.xlsx"]
df_path = next((p for p in df_candidates if p.exists()), None)
assert df_path is not None, f"No se encontró df_test_train_v8.xlsx en: {df_candidates}"
df = pd.read_excel(df_path)

```

```python
# -------- 1) Variables preferidas + alias ----------
XGB_VARS_PREF = [
    "Presupuesto_licitacion_lote_c","Plazo_m","Criterio_precio_p","N_CPV",
    "N_clasi_empresa","Mes_lici","Intervalo_lici_d_c","N_lotes",
    "Tipo_de_Administracion_c","Tipo_de_procedimiento_c","C_juicio_valor_p_c",
    "Provincia2"
]
ALIASES = {
    "Plazo_m": ["Plazo_m_c"],
    "Criterio_precio_p": ["C_precio_p"],
}

def resolve_vars(preferred, df_cols, aliases):
    used, mapping, missing = [], {}, []
    for v in preferred:
        cands = [v] + aliases.get(v, [])
        found = next((c for c in cands if c in df_cols), None)
        if found is None:
            missing.append(v)
        else:
            used.append(found); mapping[v] = found
    return used, mapping, missing

# Asegurar Provincia2 existe (derivamos si falta)
if "Provincia2" not in df.columns:
    def _to_cp_str(x):
        try: return str(int(x)).zfill(5)
        except Exception:
            s = str(x); return s if (isinstance(s, str) and s and s.lower()!="nan") else np.nan
    if "Codigo_Postal_c" in df.columns:
        df["Provincia2"] = df["Codigo_Postal_c"].apply(_to_cp_str).str[:2].astype("string")
    else:
        df["Provincia2"] = pd.Series([pd.NA]*len(df), dtype="string")

# Resolver columnas según existan en df
XGB_VARS_USED, VAR_MAP, MISSING_PREF = resolve_vars(XGB_VARS_PREF, set(df.columns), ALIASES)

# Validación de requeridos mínimos para entrenar
REQUIRED_MIN = {"Presupuesto_licitacion_lote_c","N_CPV","N_clasi_empresa","Mes_lici",
                "Intervalo_lici_d_c","N_lotes","Tipo_de_Administracion_c",
                "Tipo_de_procedimiento_c","C_juicio_valor_p_c"}
missing_min = [v for v in REQUIRED_MIN if (v not in VAR_MAP and v not in XGB_VARS_USED)]
if "N_ofertantes" not in df.columns:
    raise ValueError("❌ 'N_ofertantes' no está en df. No se puede entrenar XGB.")
if missing_min:
    raise ValueError(f"❌ Faltan columnas mínimas para XGB (tras alias): {missing_min}")

# -------- 2) Target -> clases 0/1/2 ----------
def cat_n_ofertantes(n):
    if pd.isna(n): return np.nan
    n = int(n)
    if n <= 4:  return 0
    if n <= 11: return 1
    return 2

mask = df["N_ofertantes"].notna()
X_raw = df.loc[mask, XGB_VARS_USED].copy()
y = df.loc[mask, "N_ofertantes"].apply(cat_n_ofertantes).astype(int)
labels_txt = ["BAJO","MEDIO","ALTO"]

# -------- 3) Split 80/20 estratificado ----------
X_train, X_test, y_train, y_test = train_test_split(
    X_raw, y, test_size=0.20, random_state=RANDOM_STATE, stratify=y
)

# -------- 4) Encoding (LabelEncoder) en 3 categóricas ----------
cat_vars_pref = ["Tipo_de_Administracion_c","Tipo_de_procedimiento_c","Provincia2"]
```

```python
# Mapeo preferida->usada para encoders
encoders = {}
for var_pref in cat_vars_pref:
    var_used = VAR_MAP.get(var_pref, var_pref)
    if var_used in X_train.columns:
        all_cats = pd.concat([X_train[var_used], X_test[var_used]]).astype(str).unique()
        le = LabelEncoder().fit(all_cats)
        X_train[var_used] = le.transform(X_train[var_used].astype(str))
        X_test[var_used]  = le.transform(X_test[var_used].astype(str))
        encoders[var_used] = le  # guardamos encoder con el NOMBRE REAL usado

# Mes_lici numérica
mes_col = VAR_MAP.get("Mes_lici", "Mes_lici")
if mes_col in X_train.columns and not pd.api.types.is_numeric_dtype(X_train[mes_col]):
    X_train[mes_col] = pd.to_numeric(X_train[mes_col], errors="coerce").fillna(0).astype(int)
    X_test[mes_col]  = pd.to_numeric(X_test[mes_col],  errors="coerce").fillna(0).astype(int)

# -------- 5) Modelo XGB ----------
xgb_model = xgb.XGBClassifier(
    n_estimators=50, max_depth=4, learning_rate=0.2,
    random_state=RANDOM_STATE, eval_metric="mlogloss", verbosity=0
)
xgb_model.fit(X_train, y_train)

# -------- 6) Métricas ----------
y_pred_tr = xgb_model.predict(X_train)
y_pred_te = xgb_model.predict(X_test)
p_te = xgb_model.predict_proba(X_test)

acc_tr = accuracy_score(y_train, y_pred_tr)
acc_te = accuracy_score(y_test,  y_pred_te)
f1_per = f1_score(y_test, y_pred_te, average=None)
rec_per = recall_score(y_test, y_pred_te, average=None)
f1_w   = f1_score(y_test, y_pred_te, average="weighted")

# -------- 7) Matriz de confusión ----------
cm = confusion_matrix(y_test, y_pred_te, labels=[0,1,2])
plt.figure(figsize=(6,5))
plt.imshow(cm, interpolation="nearest")
plt.title("Matriz de Confusión — XGB N_ofertantes")
plt.colorbar()
ticks = np.arange(3); plt.xticks(ticks, labels_txt, rotation=45); plt.yticks(ticks, labels_txt)
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, format(cm[i,j], "d"), ha="center", va="center")
plt.ylabel("Reales"); plt.xlabel("Predicciones")
plt.tight_layout(); plt.savefig(vdir/"cm_xgb.png", dpi=150); plt.close()

# -------- 8) Tabla de métricas por clase + total ----------
tbl = []
for i, L in enumerate(labels_txt):
    casos_reales    = int((y_test==i).sum())
    casos_predichos = int((y_pred_te==i).sum())
    prec = (cm[i,i]/casos_predichos) if casos_predichos>0 else 0.0
    tbl.append({"Clase": L, "Casos_Reales": casos_reales, "Casos_Predichos": casos_predichos,
                "F1_Score": float(f1_per[i]), "Recall": float(rec_per[i]), "Precision": float(prec)})
tbl.append({"Clase": "TOTAL", "Casos_Reales": int(len(y_test)), "Casos_Predichos": int(len(y_test)),
            "F1_Score": float(f1_w), "Recall": float(acc_te), "Precision": float(acc_te)})
pd.DataFrame(tbl).to_csv(vdir/"tabla_metricas_XGB_v8.csv", index=False)

# -------- 9) REAL vs PREDICHO + probas ----------
df_pred = pd.DataFrame({
    "ID_Test": np.arange(len(y_test)),
    "Real": y_test.values,
    "Real_Label": [labels_txt[i] for i in y_test.values],
    "Predicho": y_pred_te,
    "Predicho_Label": [labels_txt[i] for i in y_pred_te],
    "Correcto": (y_test.values == y_pred_te),
    "Prob_BAJO": p_te[:,0], "Prob_MEDIO": p_te[:,1], "Prob_ALTO": p_te[:,2],
    "Prob_Max": p_te.max(axis=1)
})
df_pred.to_csv(vdir/"real_vs_predicho_XGB_v8.csv", index=False)

# -------- 10) Datasets train/test codificados (traza) ----------
df_train_final = pd.DataFrame(X_train, columns=X_train.columns)
df_train_final["N_ofertantes_cat"]   = y_train.values
df_train_final["N_ofertantes_label"] = ["BAJO" if i==0 else ("MEDIO" if i==1 else "ALTO") for i in y_train.values]
df_test_final = pd.DataFrame(X_test, columns=X_test.columns)
df_test_final["N_ofertantes_cat"]   = y_test.values
df_test_final["N_ofertantes_label"] = ["BAJO" if i==0 else ("MEDIO" if i==1 else "ALTO") for i in y_test.values]
df_train_final.to_csv(vdir/"df_train_XGB_v8.csv", index=False)
df_test_final.to_csv(vdir/"df_test_XGB_v8.csv", index=False)

# -------- 11) Importancias ----------
imp = getattr(xgb_model, "feature_importances_", None)
if imp is not None and len(imp) == df_train_final.shape[1]-2:
    pd.DataFrame({"Variable": df_train_final.columns[:-2], "Importancia": imp})\
      .sort_values("Importancia", ascending=False)\
      .to_csv(vdir/"importancia_variables_XGB_v8.csv", index=False)

# -------- 12) Pack para PRODUCCIÓN ----------
pack = {
    "modelo": xgb_model,
    "encoders": encoders,                    # dict: {col_usada: LabelEncoder}
    "variables_predictoras": XGB_VARS_USED,  # NOMBRES REALES usados (tras alias)
    "labels": ["BAJO","MEDIO","ALTO"],
    "alias_mapping": VAR_MAP,                # mapping preferida->usada (traza)
    "metricas_test": {
        "accuracy": float(acc_te),
        "f1_scores": [float(x) for x in f1_per],
        "recall_scores": [float(x) for x in rec_per],
        "f1_weighted": float(f1_w)
    },
    "version": "v8",
    "random_state": int(RANDOM_STATE)
}
joblib.dump(pack, DIR_MODELOS / "model_Noferta_XGB_v8.pkl")

# -------- 13) Predict de ejemplo ----------
ejemplo = X_test.iloc[:1].copy()
ejemplo.to_csv(vdir/"predict_XGB_v8.csv", index=False)

# -------- 14) Resumen + explicación ----------
with open(vdir/"metrics.json","w",encoding="utf-8") as f:
    json.dump({"accuracy_train": float(acc_tr), "accuracy_test": float(acc_te),
               "f1_weighted": float(f1_w),
               "f1_per_class": dict(zip(["BAJO","MEDIO","ALTO"], [float(x) for x in f1_per])),
               "recall_per_class": dict(zip(["BAJO","MEDIO","ALTO"], [float(x) for x in rec_per]))},
              f, indent=2, ensure_ascii=False)

with open(vdir/"explicacion_tecnica.txt","w",encoding="utf-8") as f:
    f.write(
        "XGB multiclase (BAJO/MEDIO/ALTO) para N_ofertantes con split 80/20 (rs=42). "
        "Resolución de alias (p.ej., Criterio_precio_p/C_precio_p; Plazo_m/Plazo_m_c). "
        "Encoding: LabelEncoder en Tipo_de_Administracion_c, Tipo_de_procedimiento_c, Provincia2 (usando nombres reales tras alias). "
        "Mes_lici numérica. Pack de producción guarda variables usadas y el mapping de alias."
    )

print("✅ XGB N_ofertantes entrenado y guardado")
```

```python
print("→ modelos/: model_Noferta_XGB_v8.pkl")
print("→ validacion/03_xgb_n_ofertantes/: metrics.json, cm_xgb.png, tabla_metricas_XGB_v8.csv, real_vs_predicho_XGB_v8.csv, df_train_XGB_v8.csv, df_test_XGB_v8.csv, predict_XGB_v8.csv, importancia_variables_XGB_v8.csv")

"""Celda C — META del XGB (baja_comp): políticas y fallbacks"""

# ===========================================
# META — XGB baja_comp (políticas de inputs)
# ===========================================
import json, joblib, numpy as np, pandas as pd
from pathlib import Path

BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE / "modelos"
DIR_INPUTS  = BASE / "inputs"

pack_path = DIR_MODELOS / "model_Noferta_XGB_v8.pkl"
assert pack_path.exists(), "Falta modelos/model_Noferta_XGB_v8.pkl"
pack = joblib.load(pack_path)

# Cargar df de partida para medir modas (preferencia inputs/df_test_train_v8.xlsx)
df_path = None
for p in [DIR_INPUTS/"df_test_train_v8.xlsx", BASE/"df_test_train_v8.xlsx"]:
    if p.exists(): df_path = p; break
assert df_path is not None, "No encuentro df_test_train_v8.xlsx para extraer modas de categóricas."
df_all = pd.read_excel(df_path)

# Columnas y encoders reales del pack
xgb_vars  = list(pack.get("variables_predictoras", []))
xgb_encs  = pack.get("encoders", {})  # {col_usada: LabelEncoder}

# Mapa fallback por columna categórica = moda en df_all (si no está, usamos primer class_ del encoder)
fallback_class_por_col = {}
for col, le in xgb_encs.items():
    if col in df_all.columns:
        s = df_all[col].astype(str)
        if len(s.dropna()) > 0:
            moda = s.value_counts(dropna=True).idxmax()
            # Si la moda no está en el encoder (variantes), elegimos la clase más frecuente del encoder
            if moda not in set(map(str, le.classes_)):
                # fallback a la clase con más ocurrencias en df_all entre las classes_
                vc = {cls: int((s == str(cls)).sum()) for cls in le.classes_}
                moda = max(vc, key=vc.get)
            fallback_class_por_col[col] = str(moda)
        else:
            fallback_class_por_col[col] = str(le.classes_[0])
    else:
        fallback_class_por_col[col] = str(le.classes_[0])

xgb_meta = {
    "artifact": "model_Noferta_XGB_v8.pkl",
    "variables_predictoras": xgb_vars,
    "encoders_columns": list(xgb_encs.keys()),
    "labels": list(pack.get("labels", [])),
    "policies": {
        "missing_numeric_policy": "fill_0",
        "cat_unknown_policy": "fallback_to_mode",  # usar moda por columna
        "fallback_class_por_col": fallback_class_por_col,
        "threshold_bajo": 0.60
    },
    "version": pack.get("version", "v8"),
    "random_state": int(pack.get("random_state", 42))
}
(DIR_MODELOS / "xgb_meta.json").write_text(json.dumps(xgb_meta, indent=2, ensure_ascii=False), encoding="utf-8")
print("✅ Guardado modelos/xgb_meta.json")

"""RF base (PRE mínimas) — desarrollo + validación + guardados"""
```

```python

# ============================================================
# RF base (PRE mínimas) — desarrollo + validación + guardados
# ============================================================
import os, json, joblib, numpy as np, pandas as pd, matplotlib.pyplot as plt
from pathlib import Path
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix

# --- Rutas base ---
BASE_DIR    = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE_DIR / "modelos"
DIR_VALID   = BASE_DIR / "validacion"
DIR_MODELOS.mkdir(parents=True, exist_ok=True)

vdir = DIR_VALID / "04_rf_base"
vdir.mkdir(parents=True, exist_ok=True)

RANDOM_STATE = 42

# --- Cargar df y splits TRAIN/TEST ---
df_candidates = [BASE_DIR/"inputs/df_test_train_v8.xlsx", BASE_DIR/"df_test_train_v8.xlsx"]
df_path = next((p for p in df_candidates if p.exists()), None)
assert df_path is not None, f"No se encontró df_test_train_v8.xlsx en: {df_candidates}"
df = pd.read_excel(df_path)

assert "Cluster_6" in df.columns, "❌ Falta 'Cluster_6'."
assert "Dataset"   in df.columns, "❌ Falta 'Dataset' (TRAIN/TEST)."
df["Cluster_6"] = df["Cluster_6"].astype(str)
df["Dataset"]   = df["Dataset"].astype(str).str.upper()

df_train = df[df["Dataset"] == "TRAIN"].copy()
df_test  = df[df["Dataset"] == "TEST"].copy()

# 1) Variables PRE mínimas acordadas (sin Provincia2) + alias
MIN_PRE_PREF = [
    "Presupuesto_licitacion_lote_c","Plazo_m","Criterio_precio_p","C_precio_p",
    "N_CPV","N_clasi_empresa","Mes_lici","Intervalo_lici_d_c","N_lotes",
    "Tipo_de_Administracion_c","Tipo_de_procedimiento_c","C_juicio_valor_p_c"
]
ALIASES = {
    "Criterio_precio_p": ["C_precio_p"],  # preferimos Criterio_precio_p; si no, C_precio_p
    "Plazo_m": ["Plazo_m_c"],             # preferimos Plazo_m; si no, Plazo_m_c
}

def resolve_vars(preferred, df_cols, aliases):
    used = []
    mapping = {}
    # Regla especial para la variable de precio: elegir UNA (preferida si existe)
    precio_chosen = None
    for v in preferred:
        cands = [v] + aliases.get(v, [])
        found = next((c for c in cands if c in df_cols), None)
        if found is None:
            continue
        if v in ["Criterio_precio_p", "C_precio_p"]:
            if precio_chosen is not None:
                # ya elegimos una de las dos, saltamos la otra
                continue
            # si existe Criterio_precio_p, tomamos esa y marcamos; si no, tomamos C_precio_p
            if "Criterio_precio_p" in df_cols:
                precio_chosen = "Criterio_precio_p"
                used.append("Criterio_precio_p")
                mapping["Criterio_precio_p"] = "Criterio_precio_p"
            elif "C_precio_p" in df_cols:
                precio_chosen = "C_precio_p"
                used.append("C_precio_p")
                mapping["Criterio_precio_p"] = "C_precio_p"  # mapping a la usada real
            continue
        # Plazo_m vs Plazo_m_c: elegimos una
        if v == "Plazo_m":
            if "Plazo_m" in df_cols:
                used.append("Plazo_m"); mapping["Plazo_m"] = "Plazo_m"
            elif "Plazo_m_c" in df_cols:
                used.append("Plazo_m_c"); mapping["Plazo_m"] = "Plazo_m_c"
            continue
        used.append(found); mapping[v] = found
    # quitar duplicados conservando orden
    seen=set(); used_uniq=[]
    for c in used:
        if c not in seen:
            seen.add(c); used_uniq.append(c)
    return used_uniq, mapping

MIN_PRE_USED, VAR_MAP = resolve_vars(MIN_PRE_PREF, set(df.columns), ALIASES)
assert len(MIN_PRE_USED) > 0, "✕ No se resolvieron variables PRE mínimas."

# 2) Preparar X/y (TRAIN/TEST) — asegurar Mes_lici categórica para RF
Xtr = df_train[MIN_PRE_USED].copy()
Xte = df_test[MIN_PRE_USED].copy()
ytr = df_train["Cluster_6"].astype(str).copy()
yte = df_test["Cluster_6"].astype(str).copy()

mes_col = VAR_MAP.get("Mes_lici", "Mes_lici")
if mes_col in Xtr.columns:
    Xtr[mes_col] = Xtr[mes_col].astype("Int64").astype("category")
    Xte[mes_col] = Xte[mes_col].astype("Int64").astype("category")

# 3) Preprocesamiento + modelo
try:
    oh = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    oh = OneHotEncoder(handle_unknown="ignore", sparse=False)

num_cols = Xtr.select_dtypes(include=[np.number]).columns.tolist()
cat_cols = [c for c in Xtr.columns if c not in num_cols]

prep = ColumnTransformer([
    ("num", Pipeline([("imp", SimpleImputer(strategy="median")), ("sc", StandardScaler())]), num_cols),
    ("cat", Pipeline([("imp", SimpleImputer(strategy="most_frequent")), ("oh", oh)]), cat_cols),
])

rf = RandomForestClassifier(random_state=RANDOM_STATE, n_jobs=-1)
pipe = Pipeline([("prep", prep), ("clf", rf)])

# 4) Tuning ligero
param_grid = {
    "clf__n_estimators": [600, 900, 1100],
    "clf__max_depth": [None, 18, 24],
    "clf__min_samples_split": [2, 5, 10],
    "clf__min_samples_leaf": [1, 2, 4],
    "clf__max_features": ["sqrt", 0.5],
    "clf__class_weight": [None, "balanced_subsample"]
}
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_STATE)
rs = RandomizedSearchCV(
    pipe, param_distributions=param_grid, n_iter=28,
    cv=cv, scoring="f1_macro", random_state=RANDOM_STATE, n_jobs=-1, verbose=0
)
rs.fit(Xtr, ytr)
rf_base = rs.best_estimator_

# 5) Predicciones y métricas
```

```python
pred = rf_base.predict(Xte)
proba = rf_base.predict_proba(Xte)
acc = accuracy_score(yte, pred)
f1m = f1_score(yte, pred, average="macro")

def per_class_acc(y_true, y_pred):
    out={}
    for cl in sorted(pd.unique(y_true), key=lambda s:int(s)):
        m = (y_true==cl)
        out[cl] = float((y_pred[m]==y_true[m]).mean()) if m.any() else np.nan
    return out

acc_c = per_class_acc(yte, pred)

# 6) Guardados
joblib.dump(rf_base, DIR_MODELOS / "rf_base.pkl")
pd.DataFrame({"y_true": yte, "y_pred": pred}).to_csv(vdir / "pred_test_rf_base.csv", index=False)
pd.Series(acc_c).to_csv(vdir / "accuracy_por_clase.csv", header=["accuracy"])
with open(vdir / "summary.json","w",encoding="utf-8") as f:
    json.dump({
        "accuracy": float(acc),
        "macro_f1": float(f1m),
        "features_used": MIN_PRE_USED,
        "alias_mapping": VAR_MAP,
        "best_params": rs.best_params_
    }, f, indent=2, ensure_ascii=False)

# Matriz de confusión
labs = sorted(pd.unique(pd.concat([ytr, yte])), key=lambda s:int(s))
cm = confusion_matrix(yte, pred, labels=labs)
plt.figure(figsize=(6,5))
plt.imshow(cm, interpolation="nearest")
plt.title("Matriz de Confusión — RF base")
plt.colorbar()
ticks=np.arange(len(labs)); plt.xticks(ticks,labs,rotation=45, ha="right"); plt.yticks(ticks,labs)
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j,i,format(cm[i,j],'d'),ha="center",va="center")
plt.ylabel("Reales"); plt.xlabel("Predicciones")
plt.tight_layout(); plt.savefig(vdir / "cm_rf_base.png", dpi=150); plt.close()

with open(vdir / "explicacion_tecnica.txt","w",encoding="utf-8") as f:
    f.write(
        "RF base entrenado SOLO con variables PRE mínimas (sin Provincia2). "
        "Pipeline: imputación (mediana/moda), OHE(handle_unknown='ignore'), escalado numérico (var≈1). "
        "Tuning ligero con RandomizedSearchCV (f1_macro, rs=42). Artefacto final en /modelos/rf_base.pkl."
    )

print("✅ RF base entrenado y guardado en modelos/rf_base.pkl")
print(f"[TEST] Acc={acc:.4f} | Macro-F1={f1m:.4f} | Vars usadas={len(MIN_PRE_USED)}")

"""Celda D — META del RF base (features crudas exactas + dtypes esperados)"""

# ==============================================
# META — RF base (contrato de features)
# ==============================================
import json, joblib, sklearn
from pathlib import Path

BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE / "modelos"

rf_path = DIR_MODELOS / "rf_base.pkl"
assert rf_path.exists(), "Falta modelos/rf_base.pkl"
rf_base = joblib.load(rf_path)

# Extraer columnas crudas esperadas por el ColumnTransformer
def _cols_from_pipe(pipe):
    try:
        trfs = pipe.named_steps["prep"].transformers_
        cols = {}
        for name, _, c in pipe.named_steps["prep"].transformers_:
            cols[name] = list(c) if isinstance(c, list) else []
        return cols
    except Exception:
        return {"num": [], "cat": []}

cols = _cols_from_pipe(rf_base)
alias_mapping = globals().get("VAR_MAP", {})  # si existe en el notebook, lo aprovechamos

rf_meta = {
    "artifact": "rf_base.pkl",
    "expected_raw_features": cols,
    "alias_mapping_used_in_train": alias_mapping,
    "dtypes_expect": {"Mes_lici": "category"},
    "sklearn_version": sklearn.__version__
}
(DIR_MODELOS / "rf_base_meta.json").write_text(json.dumps(rf_meta, indent=2, ensure_ascii=False), encoding="utf-8")
print("✅ Guardado modelos/rf_base_meta.json")

"""RF_124_BASE+BC (train) + Ensemble “seguro” BASE+BC (test)"""

# =====================================================================
# Celda única — RF_124_BASE+BC (train) + Ensemble “seguro” BASE+BC (test)
#   - Entrena especialista 1/2/4 con MIN_PRE + baja_comp (XGB) duplicada (k=5)
#   - Construye ensemble seguro vs RF base y genera la gráfica final
#   - Guarda artefactos de PRODUCCIÓN (rf_124_basebc.pkl + ensemble_meta.json)
#   - Guarda validaciones (CSV/PNG/JSON)
# =====================================================================
import os, json, joblib, numpy as np, pandas as pd, matplotlib.pyplot as plt
from pathlib import Path
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix

# ----------------- Configuración -----------------
BASE_DIR    = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE_DIR / "modelos"
DIR_VALID   = BASE_DIR / "validacion"
DIR_MODELOS.mkdir(parents=True, exist_ok=True)
(DIR_VALID / "05_rf_124_basebc").mkdir(parents=True, exist_ok=True)
(DIR_VALID / "06_ensemble_basebc").mkdir(parents=True, exist_ok=True)

RANDOM_STATE   = 42
CLASSES_124    = {"1","2","4"}
BAJA_COMP_DUP  = 5          # ← regla acordada (k=5)
TH_BAJO        = 0.55
ALLOWED_DROP_1 = 0.05       # ← máx 5 pp
REQUIRE_IMPROVE_4 = True
REQUIRE_IMPROVE_GLOBAL = True
MARGIN_MIN     = 0.03

vdir_model  = DIR_VALID / "05_rf_124_basebc"
vdir_ens    = DIR_VALID / "06_ensemble_basebc"

# ----------------- 0) Cargar datos y prerequisitos -----------------
df_candidates = [BASE_DIR/"inputs/df_test_train_v8.xlsx", BASE_DIR/"df_test_train_v8.xlsx"]
df_path = next((p for p in df_candidates if p.exists()), None)
assert df_path is not None, f"No se encontró df_test_train_v8.xlsx en: {df_candidates}"
df = pd.read_excel(df_path)

assert "Cluster_6" in df.columns, "❌ Falta 'Cluster_6'."
```

```python
assert "Dataset"   in df.columns, "❌ Falta 'Dataset' (TRAIN/TEST)."
df["Cluster_6"] = df["Cluster_6"].astype(str)
df["Dataset"]   = df["Dataset"].astype(str).str.upper()
df_train = df[df["Dataset"] == "TRAIN"].copy()
df_test  = df[df["Dataset"] == "TEST"].copy()

rf_base_path   = DIR_MODELOS / "rf_base.pkl"
xgb_pack_path  = DIR_MODELOS / "model_Noferta_XGB_v8.pkl"
assert rf_base_path.exists(), f"❌ Falta {rf_base_path}. Entrena antes RF base."
assert xgb_pack_path.exists(), f"❌ Falta {xgb_pack_path}. Entrena antes XGB N_ofertantes."
rf_base  = joblib.load(rf_base_path)
xgb_pack = joblib.load(xgb_pack_path)

# ------------------ 1) Utilidades ------------------
MIN_PRE_PREF = [
    "Presupuesto_licitacion_lote_c","Plazo_m","Criterio_precio_p","C_precio_p",
    "N_CPV","N_clasi_empresa","Mes_lici","Intervalo_lici_d_c","N_lotes",
    "Tipo_de_Administracion_c","Tipo_de_procedimiento_c","C_juicio_valor_p_c"
]
ALIASES = {"Criterio_precio_p":["C_precio_p"], "Plazo_m":["Plazo_m_c"]}

def resolve_vars(preferred, df_cols, aliases):
    used, mapping = [], {}
    precio_chosen = None
    for v in preferred:
        cands = [v] + aliases.get(v, [])
        found = next((c for c in cands if c in df_cols), None)
        if found is None:
            continue
        if v in ["Criterio_precio_p", "C_precio_p"]:
            if precio_chosen is not None:
                continue
            if "Criterio_precio_p" in df_cols:
                precio_chosen = "Criterio_precio_p"
                used.append("Criterio_precio_p"); mapping["Criterio_precio_p"] = "Criterio_precio_p"
            elif "C_precio_p" in df_cols:
                precio_chosen = "C_precio_p"
                used.append("C_precio_p"); mapping["Criterio_precio_p"] = "C_precio_p"
            continue
        if v == "Plazo_m":
            if "Plazo_m" in df_cols:
                used.append("Plazo_m"); mapping["Plazo_m"] = "Plazo_m"; continue
            if "Plazo_m_c" in df_cols:
                used.append("Plazo_m_c"); mapping["Plazo_m"] = "Plazo_m_c"; continue
        used.append(found); mapping[v] = found
    # único conservando orden
    seen=set(); used_uniq=[]
    for c in used:
        if c not in seen:
            seen.add(c); used_uniq.append(c)
    return used_uniq, mapping

MIN_PRE_USED, VAR_MAP = resolve_vars(MIN_PRE_PREF, set(df_train.columns), ALIASES)

# Provincia2 para XGB si falta
def _to_cp_str(x):
    try: return str(int(x)).zfill(5)
    except Exception:
        s = str(x); return s if (isinstance(s,str) and s and s.lower()!="nan") else np.nan
if "Provincia2" not in df.columns:
    if "Codigo_Postal_c" in df.columns:
        df["Provincia2"] = df["Codigo_Postal_c"].apply(_to_cp_str).str[:2].astype("string")
    else:
        df["Provincia2"] = pd.Series([pd.NA]*len(df), dtype="string")
# (recalcular train/test por si añadimos la columna)
df_train = df[df["Dataset"] == "TRAIN"].copy()
df_test  = df[df["Dataset"] == "TEST"].copy()
```

```python
# Pack XGB → predictor de baja_comp (BAJO/NO BAJO con umbral 0.55)
xgb_model  = xgb_pack["modelo"]
xgb_vars   = xgb_pack["variables_predictoras"]  # nombres REALES usados en el pack
xgb_encs   = xgb_pack["encoders"]               # {col_usada: LabelEncoder}
xgb_labels = list(xgb_pack["labels"])
idx_bajo   = xgb_labels.index("BAJO") if "BAJO" in xgb_labels else 0

def comp_predict_bajo(df_in: pd.DataFrame) -> pd.Series:
    Xp = pd.DataFrame(index=df_in.index)
    for col in xgb_vars:
        Xp[col] = df_in[col] if col in df_in.columns else 0
    for col, le in xgb_encs.items():
        if col in Xp.columns:
            Xp[col] = le.transform(Xp[col].astype(str))
        else:
            Xp[col] = 0
    if "Mes_lici" in Xp.columns and not pd.api.types.is_numeric_dtype(Xp["Mes_lici"]):
        Xp["Mes_lici"] = pd.to_numeric(Xp["Mes_lici"], errors="coerce").fillna(0).astype(int)
    p = xgb_model.predict_proba(Xp[xgb_vars])[:, idx_bajo]
    return pd.Series((p >= TH_BAJO).astype(np.int8), index=df_in.index, name="baja_comp")

# Dataset BASE+BC (MIN_PRE + baja_comp categórica duplicada k veces)
def build_basebc(df_in: pd.DataFrame) -> pd.DataFrame:
    X = df_in[MIN_PRE_USED].copy()
    mes_col = VAR_MAP.get("Mes_lici","Mes_lici")
    if mes_col in X.columns:
        X[mes_col] = X[mes_col].astype("Int64").astype("category")
    bc = comp_predict_bajo(df_in).astype("category")
    X["baja_comp"] = bc
    for k in range(1, BAJA_COMP_DUP+1):
        X[f"baja_comp_dup{k}"] = X["baja_comp"]
    return X

# ------------------ 2) Preparar datos especialista 1/2/4 ------------------
Xtr_basebc = build_basebc(df_train)
Xte_basebc = build_basebc(df_test)
pd.concat([df_train[["Cluster_6"]], Xtr_basebc], axis=1).to_csv(vdir_model/"train_BASEBC_124.csv", index=False)
pd.concat([df_test[["Cluster_6"]],  Xte_basebc],  axis=1).to_csv(vdir_model/"test_BASEBC_124.csv",  index=False)

mask_tr = df_train["Cluster_6"].isin(CLASSES_124)
X124_tr  = Xtr_basebc.loc[mask_tr].copy()
y124_tr  = df_train.loc[mask_tr, "Cluster_6"].astype(str).copy()

# ------------------ 3) Entrenar RF_124 BASE+BC ------------------
num_cols = X124_tr.select_dtypes(include=[np.number]).columns.tolist()
cat_cols = [c for c in X124_tr.columns if c not in num_cols]
try:
    oh = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    oh = OneHotEncoder(handle_unknown="ignore", sparse=False)

prep = ColumnTransformer([
    ("num", Pipeline([("imp", SimpleImputer(strategy="median")), ("sc", StandardScaler())]), num_cols),
    ("cat", Pipeline([("imp", SimpleImputer(strategy="most_frequent")), ("oh", oh)]), cat_cols),
])
rf = RandomForestClassifier(random_state=RANDOM_STATE, n_jobs=-1)
pipe = Pipeline([("prep", prep), ("clf", rf)])

param_grid = {
    "clf__n_estimators":[600,900,1100],
    "clf__max_depth":[None,18,24],
    "clf__min_samples_split":[2,5,10],
    "clf__min_samples_leaf":[1,2,4],
    "clf__max_features":["sqrt",0.5],
    "clf__class_weight":[None,"balanced_subsample"]
}
```

```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_STATE)
rs = RandomizedSearchCV(pipe, param_distributions=param_grid, n_iter=28, cv=cv,
                        scoring="f1_macro", random_state=RANDOM_STATE, n_jobs=-1,
                        verbose=0)
rs.fit(X124_tr, y124_tr)
rf_124_basebc = rs.best_estimator_
joblib.dump(rf_124_basebc, DIR_MODELOS / "rf_124_basebc.pkl")
print("💾 Guardado: modelos/rf_124_basebc.pkl")

# ------------------ 4) Predicciones base y especialista en TEST ------------------
# A) RF base
def required_cols_from_pipe(pipe):
    try:
        trfs = pipe.named_steps["prep"].transformers_
        cols = []
        for _,_, cols_i in trfs:
            if isinstance(cols_i, list): cols.extend(cols_i)
        # único preservando orden
        out = []
        for c in cols:
            if c not in out: out.append(c)
        return out
    except Exception:
        return None

req_base_cols = required_cols_from_pipe(rf_base) or [c for c in df_test.columns if c
!= "Cluster_6"]
Xte_base = df_test[[c for c in req_base_cols if c in df_test.columns]].copy()
if "Mes_lici" in Xte_base.columns:
    Xte_base["Mes_lici"] = Xte_base["Mes_lici"].astype("Int64").astype("category")

proba_base = rf_base.predict_proba(Xte_base)
labels_base = [str(c) for c in rf_base.named_steps["clf"].classes_]
pred_base = np.array([labels_base[i] for i in np.argmax(proba_base, axis=1)])

# B) Especialista BASE+BC
proba_bb = rf_124_basebc.predict_proba(Xte_basebc)
labels_bb = [str(c) for c in rf_124_basebc.named_steps["clf"].classes_]
pred_bb = np.array([labels_bb[i] for i in np.argmax(proba_bb, axis=1)])

y_true = df_test["Cluster_6"].astype(str).to_numpy()

# ------------------ 5) Ensemble “seguro” (flips iterativos con margen)
------------------
def safe_ensemble(pred_b, proba_b, labels_b, pred_124, proba_124, labels_124):
    pred = pred_b.copy()
    scope = np.isin(pred_b, list(CLASSES_124))
    idx_b = {lb:i for i,lb in enumerate(labels_b)}
    idx_s = {lb:i for i,lb in enumerate(labels_124)}
    cands = []
    for i in np.where(scope)[0]:
        base_cls = pred_b[i]
        alt_cls  = pred_124[i]
        if alt_cls != base_cls and alt_cls in CLASSES_124 and base_cls in idx_b and
        alt_cls in idx_s:
            mb = proba_b[i][idx_b[base_cls]]
            ma = proba_124[i][idx_s[alt_cls]]
            if (ma - mb) >= MARGIN_MIN:
                cands.append((ma-mb, i, alt_cls))
    cands.sort(reverse=True)

    def acc_cls(y, yp, cls):
        m=(y==cls);
        return (yp[m]==y[m]).mean() if m.any() else np.nan

    acc0   = accuracy_score(y_true, pred_b)
    acc1_0 = acc_cls(y_true, pred_b, "1")
    acc4_0 = acc_cls(y_true, pred_b, "4")

    flipped = []
    cur = pred.copy()
    for _, i, alt in cands:
```

```python
        tmp = cur.copy()
        tmp[i] = alt
        acc   = accuracy_score(y_true, tmp)
        acc1  = acc_cls(y_true, tmp, "1")
        acc4  = acc_cls(y_true, tmp, "4")
        condG = (not REQUIRE_IMPROVE_GLOBAL) or (acc >= acc0 - 1e-12)
        cond4 = (not REQUIRE_IMPROVE_4)      or (acc4 >= acc4_0 - 1e-12)
        cond1 = (np.isnan(acc1_0) or np.isnan(acc1)) or (acc1 >= acc1_0 - ALLOWED_DROP_1 - 1e-12)
        if condG and cond4 and cond1:
            cur = tmp
            acc0, acc1_0, acc4_0 = acc, acc1, acc4
            flipped.append(i)
    return cur, flipped

ens_bb, flips_bb = safe_ensemble(pred_base, proba_base, labels_base, pred_bb, proba_bb, labels_bb)

# ----------------- 6) Métricas + gráfica final -----------------
def per_class_acc(y, yp):
    d={}
    for lb in sorted(pd.unique(y), key=lambda s:int(s)):
        m=(y==lb); d[str(lb)] = (yp[m]==y[m]).mean() if m.any() else np.nan
    return pd.Series(d)

acc_b  = accuracy_score(y_true, pred_base);  f1_b  = f1_score(y_true, pred_base, average="macro")
acc_bb = accuracy_score(y_true, ens_bb);     f1_bb = f1_score(y_true, ens_bb, average="macro")

pa_b  = per_class_acc(y_true, pred_base)
pa_bb = per_class_acc(y_true, ens_bb)

print(f"[TEST] RF base           -> Acc={acc_b:.4f} | Macro-F1={f1_b:.4f}")
print(f"[TEST] Ensemble BASE+BC  -> Acc={acc_bb:.4f} | Macro-F1={f1_bb:.4f} | flips={len(flips_bb)}")

# Gráfica final (barras por clase: base vs ensemble BASE+BC)
labs = sorted(pa_b.index, key=lambda s:int(s))
x = np.arange(len(labs)); w = 0.35
plt.figure(figsize=(11,4))
plt.bar(x - w/2, [pa_b[l]  for l in labs], width=w, label="RF base")
plt.bar(x + w/2, [pa_bb[l] for l in labs], width=w, label="Ensemble BASE+BC")
for i,l in enumerate(labs):
    for off,val in [(-w/2,pa_b[l]),(w/2,pa_bb[l])]:
        plt.text(x[i]+off, val+0.01, f"{val:.2f}", ha="center", va="bottom", fontsize=9)
plt.xticks(x, labs); plt.ylabel("Accuracy"); plt.title("Accuracy por clase — RF base vs Ensemble BASE+BC")
plt.legend(); plt.tight_layout(); plt.savefig(vdir_ens / "acc_por_clase_base_vs_ensemble.png", dpi=160); plt.close()

# Matriz de confusión del ensemble
cm = confusion_matrix(y_true, ens_bb, labels=labs)
plt.figure(figsize=(7,6)); plt.imshow(cm, interpolation='nearest'); plt.title("CM — Ensemble BASE+BC (1/2/4)"); plt.colorbar()
ticks=np.arange(len(labs)); plt.xticks(ticks, labs, rotation=45, ha='right'); plt.yticks(ticks, labs)
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]): plt.text(j,i,format(cm[i,j],'d'),ha="center",va="center")
plt.ylabel("Reales"); plt.xlabel("Predicciones")
plt.tight_layout(); plt.savefig(vdir_ens / "cm_ensemble_BASEBC.png", dpi=160); plt.close()

# ----------------- 7) Guardados (CSV/JSON) -----------------
pd.DataFrame({
    "y_true": y_true,
    "pred_rf_base": pred_base,
    "pred_ens_bb":  ens_bb,
    "changed_bb":   (pred_base!=ens_bb).astype(int)
}).to_csv(vdir_ens / "predicciones_comparadas_test.csv", index=False)
```

```python

pa_b.to_csv(vdir_ens / "per_class_rf_base.csv")
pa_bb.to_csv(vdir_ens / "per_class_ensemble_BASEBC.csv")

with open(DIR_MODELOS / "ensemble_meta.json","w",encoding="utf-8") as f:
    json.dump({
        "strategy": "rf_base_vs_rf124_basebc_flips_iterativos",
        "security_rules": {
            "global_not_worse": True,
            "class4_not_worse": True,
            "class1_drop_pp_max": 5
        },
        "margin_min": MARGIN_MIN,
        "uses": {
            "rf_base": "rf_base.pkl",
            "rf_124_basebc": "rf_124_basebc.pkl",
            "xgb_pack": "model_Noferta_XGB_v8.pkl"
        },
        "baja_comp": {"threshold": TH_BAJO, "duplicated_times": BAJA_COMP_DUP}
    }, f, indent=2, ensure_ascii=False)

with open(vdir_ens / "summary_ensemble.json","w",encoding="utf-8") as f:
    json.dump({
        "acc_rf_base": float(acc_b), "macro_f1_rf_base": float(f1_b),
        "acc_ensemble_BASEBC": float(acc_bb), "macro_f1_ensemble_BASEBC": float(f1_bb),
        "flips_BASEBC": int(len(flips_bb)),
        "ALLOWED_DROP_1": ALLOWED_DROP_1,
        "REQUIRE_IMPROVE_4": REQUIRE_IMPROVE_4,
        "REQUIRE_IMPROVE_GLOBAL": REQUIRE_IMPROVE_GLOBAL,
        "MARGIN_MIN": MARGIN_MIN,
        "baja_comp_dup": BAJA_COMP_DUP,
        "baja_comp_threshold": TH_BAJO
    }, f, indent=2, ensure_ascii=False)

with open(vdir_ens / "explicacion_tecnica.txt","w",encoding="utf-8") as f:
    f.write(
        "RF_124_BASE+BC entrenado con MIN_PRE + 'baja_comp' (XGB) duplicada k=5. "
        "Ensemble 'seguro' vs RF base con flips iterativos por margen (≥0.03) y reglas de back-off: "
        "no empeorar accuracy global ni clase 4; clase 1 no cae >5 pp. "
        "Artefactos de producción en /modelos y validaciones en /validacion/06_ensemble_basebc."
    )

print("✅ Especialista y Ensemble BASE+BC listos. Artefactos en modelos/ y validacion/06_ensemble_basebc/")

"""Celda E — META del RF_124_BASE+BC (MIN_PRE_USED + duplicadas BC exactas)"""

# ============================================
# META — RF_124_BASE+BC (contrato de features)
# ============================================
import json, joblib, sklearn
from pathlib import Path

BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE / "modelos"

rf124_path = DIR_MODELOS / "rf_124_basebc.pkl"
assert rf124_path.exists(), "Falta modelos/rf_124_basebc.pkl"
rf_124 = joblib.load(rf124_path)

def _cols_from_pipe(pipe):
    try:
        trfs = pipe.named_steps["prep"].transformers_
        cols = {}
        for name, _, c in pipe.named_steps["prep"].transformers_:
            cols[name] = list(c) if isinstance(c, list) else []
        return cols
    except Exception:
        return {"num": [], "cat": []}
```

```python

cols = _cols_from_pipe(rf_124)

# Detectar duplicadas de baja_comp exactamente como las vio el modelo
dup_names = [c for c in (cols.get("cat", []) + cols.get("num", [])) if c.startswith("baja_comp")]
dup_k = max([int(c.replace("baja_comp_dup","")) for c in dup_names if c != "baja_comp"] or [0])

alias_mapping = globals().get("VAR_MAP", {})  # si existe
rf124_meta = {
    "artifact": "rf_124_basebc.pkl",
    "expected_raw_features": cols,
    "alias_mapping_used_in_train": alias_mapping,
    "baja_comp_dup_k": dup_k,
    "baja_comp_dup_names": [c for c in dup_names if c != "baja_comp"],
    "requires_xgb_pack": "model_Noferta_XGB_v8.pkl",
    "sklearn_version": sklearn.__version__
}
(DIR_MODELOS / "rf_124_basebc_meta.json").write_text(json.dumps(rf124_meta, indent=2, ensure_ascii=False), encoding="utf-8")
print("✅ Guardado modelos/rf_124_basebc_meta.json")

"""Celda F — META del Ensemble (verificación)"""

# ============================================
# META — Ensemble (verificación y dependencias)
# ============================================
import json
from pathlib import Path

BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE / "modelos"

ens_path = DIR_MODELOS / "ensemble_meta.json"
assert ens_path.exists(), "Falta modelos/ensemble_meta.json"
ens = json.loads(ens_path.read_text(encoding="utf-8"))

ens.setdefault("depends_on", {})
ens["depends_on"].update({
    "rf_base": "rf_base.pkl",
    "rf_124_basebc": "rf_124_basebc.pkl",
    "xgb_pack": "model_Noferta_XGB_v8.pkl",
    "preprocess": "preprocess_pipeline.pkl",
    "centroids": "centroides_pre.json"
})

ens_path.write_text(json.dumps(ens, indent=2, ensure_ascii=False), encoding="utf-8")
print("✅ Ensemble meta verificado/actualizado (modelos/ensemble_meta.json)")

"""Celda G — Resumen rápido de contratos"""

# ============================================
# META — Resumen de contratos guardados
# ============================================
from pathlib import Path

BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
DIR_MODELOS = BASE / "modelos"
files = [
    "preprocess_meta.json",
    "knn_params.json",
    "knn_num_pipeline.joblib",
    "xgb_meta.json",
    "rf_base_meta.json",
    "rf_124_basebc_meta.json",
    "ensemble_meta.json"
]
print("Contratos/artefactos meta en modelos/:")
for f in files:
    p = DIR_MODELOS / f
    print(" -", f, "OK" if p.exists() else "❌")
```