

```

1  # -*- coding: utf-8 -*-
2  ### SCRIPT 15 - PYTHON
3  # PIPELINE DESARROLLO DE MODELOS DE PREDICCION, CLUSTERS, kNN Y CENTROIDES
4  #
5  # =====
6
7  # =====
8  # Celda A+C - PRE (TRAIN-only) + Centroides + POST FULL (TRAIN & ALL)
9  # =====
10 # BASE: /content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2
11 # Entradas:
12 #   - <BASE>/inputs/df_test_train_v8.xlsx
13 # Salidas (PROD):
14 #   - <BASE>/modelos/preprocess_pipeline.pkl
15 #   - <BASE>/modelos/centroides_pre.json
16 # Salidas (VALID):
17 #   - <BASE>/validacion/01_PRE_CENTROIDES/outputs/centroides_pre_train_matrix.csv
18 #   - <BASE>/validacion/01_PRE_CENTROIDES/outputs/centroides_post_train_full.csv
19 #   - <BASE>/validacion/01_PRE_CENTROIDES/outputs/centroides_post_all_full.csv
20 #   - <BASE>/validacion/01_PRE_CENTROIDES/README_STEP.txt
21
22 import os, json, joblib
23 import numpy as np
24 import pandas as pd
25 from datetime import datetime
26
27 from sklearn.compose import ColumnTransformer
28 from sklearn.pipeline import Pipeline
29 from sklearn.preprocessing import OneHotEncoder, StandardScaler
30 from sklearn.impute import SimpleImputer
31
32 # Montaje Drive
33 try:
34     from google.colab import drive # type: ignore
35     drive.mount('/content/drive')
36 except Exception:
37     pass
38
39 # === Rutas base (TODO bajo _v2) ===
40 BASE_DIR = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
41 ORIGEN_DIR = BASE_DIR
42 DIR_MODELOS = f"{BASE_DIR}/modelos"
43 DIR_VALID = f"{BASE_DIR}/validacion"
44 DIR_COMP = f"{BASE_DIR}/comparativa" # opcional (no escribimos por defecto)
45
46 STEP_TAG = "01_PRE_CENTROIDES"
47 STEP_DIR = os.path.join(DIR_VALID, STEP_TAG)
48 OUT_DIR = os.path.join(STEP_DIR, "outputs")
49 os.makedirs(DIR_MODELOS, exist_ok=True)
50 os.makedirs(OUT_DIR, exist_ok=True)
51
52 random_state = 42
53
54 print(f"📁 BASE_DIR: {BASE_DIR}")
55 print(f"📁 DIR_MODELOS: {DIR_MODELOS}")
56 print(f"📁 DIR_VALID: {DIR_VALID}")
57
58 # =====
59 # 1) Datos
60 # =====
61 df_path_candidates = [
62     f"{ORIGEN_DIR}/inputs/df_test_train_v8.xlsx",
63     f"{ORIGEN_DIR}/df_test_train_v8.xlsx",
64 ]
65 df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
66 assert df_path is not None, f"No se encontró df_test_train_v8.xlsx en: {df_path_candidates}"
67 df = pd.read_excel(df_path)
68 print(f"✓ df_test_train_v8.xlsx cargado: {df.shape}")
69
70 # Checks mínimos
71 if 'Cluster_6' not in df.columns:

```

```

72     raise ValueError("X Faltan 'Cluster_6'.")
73 if 'Dataset' not in df.columns:
74     raise ValueError("X Faltan 'Dataset' (TRAIN/TEST).")
75
76 df['Cluster_6'] = df['Cluster_6'].astype(str)
77 df['Dataset'] = df['Dataset'].astype(str).str.upper()
78
79 # =====
80 # 2) Variables PRE
81 # =====
82 pre_num = [
83     "N_lotes", "C_precio_p", "N_CPV", "N_clasi_empresa", "Presupuesto_licitacion_lote_c",
84     "Plazo_m_c", "C_juicio_valor_p_c", "Intervalo_lici_d_c",
85 ]
86 pre_cat = [
87     "Tipo_de_contrato_c", "Tipo_de_Administracion_c", "Tipo_de_procedimiento_c",
88     "Tramitacion_c",
89     # Provincia2 se añade si falta
90     "Tipo_ganador_lote_c", "Mes_lici",
91 ]
92 # Derivar Provincia2 (dos primeros dígitos)
93 def _to_cp_str(x):
94     try:
95         return str(int(x)).zfill(5)
96     except Exception:
97         s = str(x); return s if s and s.lower() != 'nan' else None
98
99 if "Provincia2" not in df.columns:
100     if "Codigo_Postal_c" in df.columns:
101         prov = df["Codigo_Postal_c"].apply(_to_cp_str).str[:2]
102         df["Provincia2"] = prov.where(prov.notna(), np.nan)
103     else:
104         df["Provincia2"] = np.nan
105 if "Provincia2" not in pre_cat:
106     pre_cat = pre_cat[:4] + ["Provincia2"] + pre_cat[4:]
107
108 # Completar columnas ausentes
109 for c in pre_num:
110     if c not in df.columns: df[c] = np.nan
111 for c in pre_cat:
112     if c not in df.columns: df[c] = np.nan
113
114 # =====
115 # 3) PREPROCESSOR (TRAIN-only)
116 # =====
117 df_train = df[df["Dataset"] == "TRAIN"].copy()
118 df_all = df.copy()
119
120 num_pipe = Pipeline([
121     ("imputer", SimpleImputer(strategy="median")),
122     ("scaler", StandardScaler())
123 ])
124 cat_pipe = Pipeline([
125     ("imputer", SimpleImputer(strategy="most_frequent")),
126     ("ohe", OneHotEncoder(handle_unknown="ignore", sparse_output=False))
127 ])
128
129 preprocessor = ColumnTransformer(
130     transformers=[("num", num_pipe, pre_num), ("cat", cat_pipe, pre_cat)],
131     remainder='drop',
132     verbose_feature_names_out=True
133 )
134
135 X_train_pre = df_train[pre_num + pre_cat].copy()
136 y_train = df_train["Cluster_6"].astype(str).copy()
137
138 preprocessor.fit(X_train_pre, y_train)
139 feature_names = preprocessor.get_feature_names_out().tolist()
140 print(f"* Features transformadas: {len(feature_names)}")
141
142 # =====

```

```

143 # 4) Centroides PRE (espacio transformado, TRAIN-only)
144 # =====
145 Z_train = preprocessor.transform(X_train_pre)
146 classes = sorted(y_train.unique(), key=lambda x: int(x))
147
148 centroids_pre, p95_intra = {}, {}
149 for c in classes:
150     Zc = Z_train[(y_train == c).values]
151     mu = Zc.mean(axis=0)
152     centroids_pre[c] = mu.tolist()
153     d = np.sqrt(((Zc - mu) ** 2).sum(axis=1))
154     p95_intra[c] = float(np.percentile(d, 95))
155
156 # Guardar artefactos (PROD)
157 joblib.dump(preprocessor, f"{DIR_MODELOS}/preprocess_pipeline.pkl")
158 with open(f"{DIR_MODELOS}/centroides_pre.json", "w", encoding="utf-8") as f:
159     json.dump({
160         "classes": classes,
161         "feature_names": feature_names,
162         "centroids_pre": centroids_pre,
163         "p95_intra": p95_intra,
164         "random_state": random_state,
165         "built_at": datetime.now().isoformat()
166     }, f, ensure_ascii=False, indent=2)
167 print("📁 Guardado (PROD): preprocess_pipeline.pkl, centroides_pre.json")
168
169 # Copias de validación
170 cent_pre_mat = pd.DataFrame.from_dict(centroids_pre, orient='index', columns=
feature_names)
171 cent_pre_mat.index.name = "Cluster_6"
172 cent_pre_mat.sort_index(key=lambda x: x.map(int), inplace=True)
173 cent_pre_mat.to_csv(f"{OUT_DIR}/centroides_pre_train_matrix.csv")
174
175 def _top12_share(series: pd.Series):
176     vc = series.value_counts(dropna=True)
177     if len(vc) == 0: return (np.nan, np.nan, np.nan, np.nan)
178     top1, s1 = vc.index[0], float(vc.iloc[0]) / float(vc.sum())
179     if len(vc) > 1:
180         top2, s2 = vc.index[1], float(vc.iloc[1]) / float(vc.sum())
181     else:
182         top2, s2 = (np.nan, np.nan)
183     return (top1, s1, top2, s2)
184
185 def _post_full(df_base: pd.DataFrame, idx_vals):
186     pre_num_present = [c for c in pre_num if c in df_base.columns]
187     post_num = [c for c in ["N_ofertantes", "Ofertas_admitidas", "Desierta",
"Importe_adjudicacion_lote"] if c in df_base.columns]
188     out = pd.DataFrame(index=idx_vals)
189     if pre_num_present:
190         g = df_base.groupby("Cluster_6")
191         pre_mean = g[pre_num_present].mean(numeric_only=True).add_prefix("PRE__").
add_suffix("__mean")
192         pre_std = g[pre_num_present].std(numeric_only=True).add_prefix("PRE__").
add_suffix("__std")
193         out = out.join(pre_mean, how="left").join(pre_std, how="left")
194     if post_num:
195         g = df_base.groupby("Cluster_6")
196         post_mean = g[post_num].mean(numeric_only=True).add_prefix("POST__").
add_suffix("__mean")
197         post_std = g[post_num].std(numeric_only=True).add_prefix("POST__").add_suffix
("__std")
198         out = out.join(post_mean, how="left").join(post_std, how="left")
199     rows = []
200     for c in idx_vals:
201         sub = df_base[df_base["Cluster_6"].astype(str) == c]
202         row = {"Cluster_6": c}
203         for col in pre_cat:
204             if col not in sub.columns:
205                 row[f"PRECAT__{col}__top1"] = np.nan; row[f"PRECAT__{col}__top1_share"] =
np.nan
206                 row[f"PRECAT__{col}__top2"] = np.nan; row[f"PRECAT__{col}__top2_share"] =
np.nan

```

```

207         else:
208             top1, s1, top2, s2 = _top12_share(sub[col])
209             row[f"PRECAT__{col}__top1"]=top1; row[f"PRECAT__{col}__top1_share"]=s1
210             row[f"PRECAT__{col}__top2"]=top2; row[f"PRECAT__{col}__top2_share"]=s2
211         rows.append(row)
212     out2 = pd.DataFrame(rows).set_index("Cluster_6")
213     final = out.join(out2, how="left")
214     final.index.name = "Cluster_6"
215     final.sort_index(key=lambda x: x.map(int), inplace=True)
216     return final
217
218 post_train = _post_full(df_train.copy(), classes)
219 post_all = _post_full(df_all.copy(), classes)
220 post_train.to_csv(f"{OUT_DIR}/centroides_post_train_full.csv")
221 post_all.to_csv(f"{OUT_DIR}/centroides_post_all_full.csv")
222
223 with open(os.path.join(STEP_DIR, "README_STEP.txt"), "w", encoding="utf-8") as f:
224     f.write(f"""STEP: {STEP_TAG}
225 BASE_DIR: {BASE_DIR}
226 Fecha: {datetime.now().isoformat()}
227 random_state: {random_state}
228 Artefactos PROD: modelos/preprocess_pipeline.pkl, modelos/centroides_pre.json
229 Salidas VALID: outputs/*.csv
230 """)
231 print("☑ Celda A+C COMPLETADA (todo bajo _v2).")
232
233 """Celda D – Distancias a centroides (TRAIN/TEST)"""
234
235 # =====
236 # Celda D – Distancias a centroides PRE (TRAIN/TEST)
237 # =====
238 # Entradas:
239 # - <BASE>/inputs/df_test_train_v8.xlsx
240 # - <BASE>/modelos/preprocess_pipeline.pkl
241 # - <BASE>/modelos/centroides_pre.json
242 # Salidas (VALID):
243 # - <BASE>/validacion/03_DISTANCIAS_PRE/outputs/df_train_with_dist.csv
244 # - <BASE>/validacion/03_DISTANCIAS_PRE/outputs/df_test_with_dist.csv
245 # - <BASE>/validacion/03_DISTANCIAS_PRE/outputs/summary_distancias.json
246
247 import os, json, joblib
248 import numpy as np
249 import pandas as pd
250 from datetime import datetime
251
252 try:
253     from google.colab import drive # type: ignore
254     drive.mount('/content/drive')
255 except Exception:
256     pass
257
258 BASE_DIR = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
259 ORIGEN_DIR = BASE_DIR
260 DIR_MODELOS = f"{BASE_DIR}/modelos"
261 DIR_VALID = f"{BASE_DIR}/validacion"
262
263 STEP_TAG = "03_DISTANCIAS_PRE"
264 STEP_DIR = os.path.join(DIR_VALID, STEP_TAG)
265 OUT_DIR = os.path.join(STEP_DIR, "outputs")
266 os.makedirs(OUT_DIR, exist_ok=True)
267
268 # Artefactos
269 preproc_path = os.path.join(DIR_MODELOS, "preprocess_pipeline.pkl")
270 cent_json_path = os.path.join(DIR_MODELOS, "centroides_pre.json")
271 assert os.path.exists(preproc_path), "Falta preprocess_pipeline.pkl"
272 assert os.path.exists(cent_json_path), "Falta centroides_pre.json"
273
274 preprocessor = joblib.load(preproc_path)
275 with open(cent_json_path, "r", encoding="utf-8") as f:
276     cent_meta = json.load(f)
277
278 classes = [str(c) for c in cent_meta["classes"]]

```

```

279 centroids_pre = {str(k): np.array(v, dtype=float) for k, v in cent_meta[
"centroids_pre"].items()}
280
281 # Datos
282 df_path_candidates = [
283     f"{ORIGEN_DIR}/inputs/df_test_train_v8.xlsx",
284     f"{ORIGEN_DIR}/df_test_train_v8.xlsx",
285 ]
286 df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
287 assert df_path is not None, "No se encontró df_test_train_v8.xlsx"
288 df = pd.read_excel(df_path)
289 if "Cluster_6" in df.columns: df["Cluster_6"] = df["Cluster_6"].astype(str)
290 df["Dataset"] = df["Dataset"].astype(str).str.upper()
291
292 pre_num = [
293     "N_lotes", "C_precio_p", "N_CPV", "N_clasi_empresa", "Presupuesto_licitacion_lote_c",
294     "Plazo_m_c", "C_juicio_valor_p_c", "Intervalo_lici_d_c",
295 ]
296 pre_cat = [
297     "Tipo_de_contrato_c", "Tipo_de_Administracion_c", "Tipo_de_procedimiento_c",
298     "Tramitacion_c",
299     "Provincia2", "Tipo_ganador_lote_c", "Mes_lici",
300 ]
301 def _to_cp_str(x):
302     try:
303         return str(int(x)).zfill(5)
304     except Exception:
305         s = str(x); return s if s and s.lower() != 'nan' else None
306
307 def _prepare_pre(df_in: pd.DataFrame) -> pd.DataFrame:
308     df = df_in.copy()
309     if "Provincia2" not in df.columns:
310         if "Codigo_Postal_c" in df.columns:
311             prov = df["Codigo_Postal_c"].apply(_to_cp_str).str[:2]
312             df["Provincia2"] = prov.where(prov.notna(), np.nan)
313         else:
314             df["Provincia2"] = np.nan
315     if "Mes_lici" not in df.columns: df["Mes_lici"] = np.nan
316     for c in pre_num:
317         if c not in df.columns: df[c] = np.nan
318     for c in pre_cat:
319         if c not in df.columns: df[c] = np.nan
320     return df[pre_num + pre_cat]
321
322 df_train = df[df["Dataset"] == "TRAIN"].copy()
323 df_test = df[df["Dataset"] == "TEST"].copy()
324 Z_train = preprocessor.transform(_prepare_pre(df_train))
325 Z_test = preprocessor.transform(_prepare_pre(df_test))
326
327 def _dist_to_centroids(Z):
328     rows=[]
329     for i in range(Z.shape[0]):
330         z=Z[i]; dists={}
331         for c in classes:
332             mu = centroids_pre[c]
333             dists[f"dist_c{c}"] = float(np.sqrt(((z - mu) ** 2).sum()))
334             items = sorted(dists.items(), key=lambda x: x[1])
335             min_k, min_d = items[0]; second_d = items[1][1] if len(items)>1 else np.nan
336             rows.append({"min_cluster": min_k.replace("dist_c", ""), "min_dist": min_d,
337                 "margin": float(second_d-min_d) if np.isfinite(second_d) else np.nan, **dists
338             })
339     return pd.DataFrame(rows)
340
341 dist_train = _dist_to_centroids(Z_train)
342 dist_test = _dist_to_centroids(Z_test)
343
344 train_out = pd.concat([df_train.reset_index(drop=True), dist_train], axis=1)
345 test_out = pd.concat([df_test.reset_index(drop=True), dist_test], axis=1)
346
347 train_out_path = os.path.join(OUT_DIR, "df_train_with_dist.csv")
348 test_out_path = os.path.join(OUT_DIR, "df_test_with_dist.csv")

```

```

347 train_out.to_csv(train_out_path, index=False)
348 test_out.to_csv(test_out_path, index=False)
349
350 with open(os.path.join(OUT_DIR, "summary_distancias.json"), "w", encoding="utf-8") as f:
351     json.dump({
352         "step": STEP_TAG,
353         "built_at": datetime.now().isoformat(),
354         "outputs": {"train_with_dist": train_out_path, "test_with_dist": test_out_path
355     }, f, ensure_ascii=False, indent=2)
356
357 with open(os.path.join(STEP_DIR, "README_STEP.txt"), "w", encoding="utf-8") as f:
358     f.write(f"{STEP_TAG} generado el {datetime.now().isoformat()} (BASE_DIR={BASE_DIR
359     })\n")
360
361 print("☑ Distancias PRE generadas (todo bajo _v2).")
362
363 """Celda A – META del PRE + centroides (preprocess_pipeline)"""
364 # =====
365 # META – PREPROCESS + CENTROIDES (guardado)
366 # =====
367 import json, joblib, numpy as np
368 from pathlib import Path
369
370 BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
371 DIR_MODELOS = BASE / "modelos"
372 DIR_MODELOS.mkdir(parents=True, exist_ok=True)
373
374 preproc_path = DIR_MODELOS / "preprocess_pipeline.pkl"
375 centroids_path = DIR_MODELOS / "centroides_pre.json"
376 assert preproc_path.exists() and centroids_path.exists(), "Faltan
377 preprocess_pipeline.pkl y/o centroides_pre.json"
378
379 pre = joblib.load(preproc_path)
380 cent = json.loads(centroids_path.read_text(encoding="utf-8"))
381
382 # Extraer columnas crudas esperadas por el ColumnTransformer
383 num_cols = []
384 cat_cols = []
385 try:
386     trfs = pre.named_transformers_
387     if "num" in trfs: num_cols = list(pre.transformers_[0][2]) if isinstance(pre.
388     transformers_[0][2], list) else []
389     if "cat" in trfs: cat_cols = list(pre.transformers_[1][2]) if isinstance(pre.
390     transformers_[1][2], list) else []
391 except Exception:
392     pass
393
394 pre_meta = {
395     "artifact": "preprocess_pipeline.pkl",
396     "expects_raw_features": {
397         "num": num_cols,
398         "cat": cat_cols
399     },
400     "transformers": {
401         "num": [s[0] for s in getattr(pre.named_transformers_.get("num", None),
402         "steps", [])] if "num" in pre.named_transformers_ else [],
403         "cat": [s[0] for s in getattr(pre.named_transformers_.get("cat", None),
404         "steps", [])] if "cat" in pre.named_transformers_ else []
405     },
406     "centroides_ref": "centroides_pre.json",
407     "centroides_overview": {
408         "classes": cent.get("classes"),
409         "p95_intra_available": bool(cent.get("p95_intra"))
410     }
411 }
412 (DIR_MODELOS / "preprocess_meta.json").write_text(json.dumps(pre_meta, indent=2,
413 ensure_ascii=False), encoding="utf-8")
414 print("☑ Guardado modelos/preprocess_meta.json")
415

```

```

410 """Celda E - kNN PRE (TRAIN y ALL) - NUM-ONLY"""
411
412 # =====
413 # Celda E - kNN de similares (TRAIN y ALL) - NUM-ONLY
414 # =====
415 # Entradas:
416 # - <BASE>/inputs/df_test_train_v8.xlsx
417 # Salidas (PROD):
418 # - <BASE>/modelos/knn_pre_train.joblib
419 # - <BASE>/modelos/knn_pre_all.joblib
420 # - <BASE>/modelos/knn_params.json
421 # Salidas (VALID):
422 # - <BASE>/validacion/04_KNN_PRE/outputs/knn_train_cases.parquet
423 # - <BASE>/validacion/04_KNN_PRE/outputs/knn_all_cases.parquet
424 # - <BASE>/validacion/04_KNN_PRE/outputs/meta_knn.json
425
426 import os, json, joblib
427 import numpy as np
428 import pandas as pd
429 from datetime import datetime
430 from sklearn.neighbors import NearestNeighbors
431 from sklearn.pipeline import Pipeline
432 from sklearn.impute import SimpleImputer
433 from sklearn.preprocessing import StandardScaler
434
435 try:
436     from google.colab import drive # type: ignore
437     drive.mount('/content/drive')
438 except Exception:
439     pass
440
441 BASE_DIR = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
442 ORIGEN_DIR = BASE_DIR
443 DIR_MODELOS = f"{BASE_DIR}/modelos"
444 DIR_VALID = f"{BASE_DIR}/validacion"
445
446 STEP_TAG = "04_KNN_PRE"
447 STEP_DIR = os.path.join(DIR_VALID, STEP_TAG)
448 OUT_DIR = os.path.join(STEP_DIR, "outputs")
449 os.makedirs(DIR_MODELOS, exist_ok=True)
450 os.makedirs(OUT_DIR, exist_ok=True)
451
452 # Datos
453 df_path_candidates = [
454     f"{ORIGEN_DIR}/inputs/df_test_train_v8.xlsx",
455     f"{ORIGEN_DIR}/df_test_train_v8.xlsx",
456 ]
457 df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
458 assert df_path is not None, "No se encontró df_test_train_v8.xlsx"
459 df = pd.read_excel(df_path)
460 df["Dataset"] = df["Dataset"].astype(str).str.upper()
461
462 df_train = df[df["Dataset"] == "TRAIN"].copy()
463 df_test = df[df["Dataset"] == "TEST"].copy()
464 df_all = pd.concat([df_train, df_test], axis=0, ignore_index=True)
465
466 # PRE NUM-ONLY
467 pre_num = [
468     "N_lotes", "C_precio_p", "N_CPV", "N_clasi_empresa", "Presupuesto_licitacion_lote_c",
469     "Plazo_m_c", "C_juicio_valor_p_c", "Intervalo_lici_d_c",
470 ]
471
472 def _prepare_num(df_in: pd.DataFrame) -> pd.DataFrame:
473     df = df_in.copy()
474     for c in pre_num:
475         if c not in df.columns: df[c] = np.nan
476         df[c] = pd.to_numeric(df[c], errors="coerce")
477     return df[pre_num]
478
479 X_train_num = _prepare_num(df_train)
480 X_all_num = _prepare_num(df_all)
481

```



```

482 num_only = Pipeline([
483     ("imputer", SimpleImputer(strategy="median")),
484     ("scaler", StandardScaler())
485 ])
486 num_only.fit(X_train_num) # TRAIN-only
487
488 Z_train = num_only.transform(X_train_num)
489 Z_all = num_only.transform(X_all_num)
490
491 # kNN + p95
492 K_NEIGH = 25
493 nn_train = NearestNeighbors(n_neighbors=K_NEIGH, metric="euclidean").fit(Z_train)
494 nn_all = NearestNeighbors(n_neighbors=K_NEIGH, metric="euclidean").fit(Z_all)
495
496 nn_tmp = NearestNeighbors(n_neighbors=2, metric="euclidean").fit(Z_train)
497 dist2, _ = nn_tmp.kneighbors(Z_train, n_neighbors=2, return_distance=True)
498 nn1_dists = dist2[:, 1]
499 p95_nn1_train = float(np.percentile(nn1_dists, 95))
500 print(f"🔍 p95_nn1_train (NUM-ONLY) = {p95_nn1_train:.4f}")
501
502 # Guardar artefactos PROD
503 joblib.dump(nn_train, os.path.join(DIR_MODELOS, "knn_pre_train.joblib"))
504 joblib.dump(nn_all, os.path.join(DIR_MODELOS, "knn_pre_all.joblib"))
505 with open(os.path.join(DIR_MODELOS, "knn_params.json"), "w", encoding="utf-8") as f:
506     json.dump({
507         "built_at": datetime.now().isoformat(),
508         "k_neighbors": K_NEIGH,
509         "p95_nn1_train": p95_nn1_train,
510         "metric": "euclidean",
511         "index_key": "knn_row",
512         "space": "pre_num",
513         "pre_num_features": pre_num
514     }, f, ensure_ascii=False, indent=2)
515 print("📁 Guardado (PROD): knn_pre_train.joblib, knn_pre_all.joblib, knn_params.json")
516
517 # Casos VALID
518 def _attach_knn_row(df_src: pd.DataFrame) -> pd.DataFrame:
519     out = df_src.reset_index(drop=True).copy()
520     out.insert(0, "knn_row", np.arange(len(out), dtype=int))
521     return out
522
523 cases_train = _attach_knn_row(df_train)
524 cases_all = _attach_knn_row(df_all)
525
526 train_cases_path = os.path.join(OUT_DIR, "knn_train_cases.parquet")
527 all_cases_path = os.path.join(OUT_DIR, "knn_all_cases.parquet")
528 cases_train.to_parquet(train_cases_path, index=False)
529 cases_all.to_parquet(all_cases_path, index=False)
530
531 with open(os.path.join(OUT_DIR, "meta_knn.json"), "w", encoding="utf-8") as f:
532     json.dump({
533         "artifacts": {"nn_train": "modelos/knn_pre_train.joblib", "nn_all":
534             "modelos/knn_pre_all.joblib"},
535         "cases": {"train": train_cases_path, "all": all_cases_path, "format":
536             "parquet", "index_key": "knn_row"},
537         "p95_nn1_train": p95_nn1_train, "space": "pre_num"
538     }, f, ensure_ascii=False, indent=2)
539
540 with open(os.path.join(STEP_DIR, "README_STEP.txt"), "w", encoding="utf-8") as f:
541     f.write(f"{STEP_TAG} (NUM-ONLY) generado el {datetime.now().isoformat()}
542         (BASE_DIR={BASE_DIR}).\n")
543
544 print("✅ kNN NUM-ONLY listo (todo bajo _v2).")
545
546 """Celda B - META del kNN (incluye pipeline num-only fitted)"""
547
548 # =====
549 # META - KNN (pipeline num-only + meta JSON)
550 # =====
551 import json, joblib
552 from pathlib import Path

```



```

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

554
555 # Guardar pipeline num-only usada realmente
556 knn_num_pipe_path = DIR_MODELOS / "knn_num_pipeline.joblib"
557 joblib.dump(num_only, knn_num_pipe_path)
558
559 # Actualizar/crear meta del KNN (aprovecha knn_params.json creado en Celda E)
560 knn_params_path = DIR_MODELOS / "knn_params.json"
561 assert knn_params_path.exists(), "Falta modelos/knn_params.json (Célula E)."
562 params = json.loads(knn_params_path.read_text(encoding="utf-8"))
563 params["num_only_pipeline_file"] = "knn_num_pipeline.joblib"
564 params["note"] = "Pipeline num-only exacta usada para estandarizar las 8 features
(NUM-ONLY) antes del índice."
565 knn_params_path.write_text(json.dumps(params, indent=2, ensure_ascii=False), encoding=
"utf-8")
566
567 print("☑ Guardado modelos/knn_num_pipeline.joblib y actualizado
modelos/knn_params.json")
568
569 """Celda F – Validación kNN (ALL vs TRAIN) con TEST"""
570
571 # =====
572 # Celda E+1 – Catálogo kNN (ALL) COMPLETO
573 # =====
574 # BASE: /content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2
575 # Entradas:
576 # - <BASE>/inputs/df_test_train_v8.xlsx
577 # - (opcional) <BASE>/validacion/04_KNN_PRE/outputs/knn_all_cases.parquet (para
verificar alineación)
578 # Salidas (PROD):
579 # - <BASE>/modelos/knn_catalog_all.parquet (TODAS las variables + knn_row)
580 # - <BASE>/modelos/knn_catalog_meta.json
581 # Salidas (VALID):
582 # - <BASE>/validacion/04_KNN_PRE/outputs/knn_catalog_all_head.csv (muestra)
583 # - <BASE>/validacion/04_KNN_PRE/outputs/knn_catalog_alignment.json (reporte
verificación)
584
585 import os, json
586 import numpy as np
587 import pandas as pd
588 from datetime import datetime
589
590 # Rutas base
591 BASE_DIR = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
592 DIR_MODELOS = f"{BASE_DIR}/modelos"
593 DIR_VALID = f"{BASE_DIR}/validacion"
594 STEP_DIR = os.path.join(DIR_VALID, "04_KNN_PRE")
595 OUT_DIR = os.path.join(STEP_DIR, "outputs")
596 os.makedirs(DIR_MODELOS, exist_ok=True)
597 os.makedirs(OUT_DIR, exist_ok=True)
598
599 # 1) Cargar df y recrear ALL (TRAIN+TEST)
600 df_path_candidates = [
601     f"{BASE_DIR}/inputs/df_test_train_v8.xlsx",
602     f"{BASE_DIR}/df_test_train_v8.xlsx",
603 ]
604 df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
605 assert df_path is not None, "No se encontró df_test_train_v8.xlsx"
606 df = pd.read_excel(df_path)
607 assert "Dataset" in df.columns, "X Falta 'Dataset' (TRAIN/TEST)"
608 df["Dataset"] = df["Dataset"].astype(str).str.upper()
609
610 df_train = df[df["Dataset"] == "TRAIN"].copy()
611 df_test = df[df["Dataset"] == "TEST"].copy()
612 df_all = pd.concat([df_train, df_test], axis=0, ignore_index=True)
613
614 # 2) Asegurar Provincia2 si falta (regla del proyecto)
615 def _to_cp_str(x):
616     try: return str(int(x)).zfill(5)

```

```

617     except Exception:
618         s = str(x); return s if (isinstance(s, str) and s and s.lower()!="nan") else
            np.nan
619
620 if "Provincia2" not in df_all.columns:
621     if "Codigo_Postal_c" in df_all.columns:
622         df_all["Provincia2"] = df_all["Codigo_Postal_c"].apply(_to_cp_str).str[:2]
623     else:
624         df_all["Provincia2"] = np.nan
625
626 # 3) Asegurar columna de ID amigable (no sobrescribimos si ya existe)
627 id_candidates = ["Identificador", "identificador", "ID", "Id", "id_lote", "id_contrato"]
628 id_col = next((c for c in id_candidates if c in df_all.columns), None)
629 if id_col is None:
630     df_all["Identificador"] = [f"row_{i}" for i in range(len(df_all))]
631     id_col = "Identificador"
632
633 # 4) Insertar knn_row para alinear con Celda E (reset_index + rango continuo)
634 catalog = df_all.reset_index(drop=True).copy()
635 catalog.insert(0, "knn_row", np.arange(len(catalog), dtype=int))
636
637 # 5) Guardar catálogo completo (PRODUCCIÓN)
638 catalog_path = os.path.join(DIR_MODELOS, "knn_catalog_all.parquet")
639 catalog.to_parquet(catalog_path, index=False)
640
641 # 6) Muestra y metadatos (VALIDACIÓN)
642 head_path = os.path.join(OUT_DIR, "knn_catalog_all_head.csv")
643 catalog.head(30).to_csv(head_path, index=False)
644
645 meta = {
646     "built_at": datetime.now().isoformat(),
647     "rows": int(len(catalog)),
648     "cols": int(catalog.shape[1]),
649     "id_col": id_col,
650     "index_key": "knn_row",
651     "source_df": os.path.basename(df_path),
652     "note": "Catálogo ALL (TRAIN+TEST) con TODAS las variables + knn_row al frente."
653 }
654 with open(os.path.join(DIR_MODELOS, "knn_catalog_meta.json"), "w", encoding="utf-8")
655 as f:
656     json.dump(meta, f, ensure_ascii=False, indent=2)
657
658 # 7) Verificación de alineación con knn_all_cases.parquet (si existe)
659 align_report = {"checked": False, "ok": None, "details": {}}
660 cases_all_path = os.path.join(OUT_DIR, "knn_all_cases.parquet")
661 if os.path.exists(cases_all_path):
662     cases_all = pd.read_parquet(cases_all_path)
663     align_report["checked"] = True
664     try:
665         # mismas filas y mismo rango de knn_row
666         same_len = (len(cases_all) == len(catalog))
667         same_range = (cases_all["knn_row"].iloc[0] == 0 and cases_all["knn_row"].iloc
668             [-1] == len(catalog)-1)
669         # chequeo de identidad en 5 posiciones aleatorias
670         rng = np.random.RandomState(42)
671         sample_idx = rng.choice(len(catalog), size=min(5, len(catalog)), replace=False)
672         id_match = []
673         for i in sample_idx:
674             r_cases = cases_all.iloc[i]
675             r_cata = catalog.iloc[i]
676             id_match.append(bool(r_cases.get("knn_row", -1) == r_cata.get("knn_row", -
677                 2)))
678         align_ok = same_len and same_range and all(id_match)
679         align_report["ok"] = bool(align_ok)
680         align_report["details"] = {
681             "same_len": bool(same_len),
682             "same_range": bool(same_range),
683             "sample_id_match_rate": float(np.mean(id_match)) if len(id_match)>0 else
684                 None
685         }
686     except Exception as e:

```

```

683         align_report["ok"] = False
684         align_report["details"] = {"error": str(e)}
685
686     with open(os.path.join(OUT_DIR, "knn_catalog_alignment.json"), "w", encoding="utf-8")
        as f:
687         json.dump(align_report, f, ensure_ascii=False, indent=2)
688
689     print("☑ Catálogo kNN ALL generado.")
690     print(f"→ modelos/knn_catalog_all.parquet | filas={len(catalog)} cols={catalog.shape[
        1]}")
691     print(f"→ validacion/04_KNN_PRE/outputs/knn_catalog_all_head.csv")
692     if align_report["checked"]:
693         print(f"✓ Verificación con knn_all_cases.parquet | ok={align_report['ok']} |
        details={align_report['details']}")
694     else:
695         print(f"❌ No se encontró knn_all_cases.parquet; se omitió verificación de
        alineación.")
696
697     """Celda E+1 – Catálogo kNN (ALL) COMPLETO"""
698
699     # =====
700     # Celda F – Validación kNN (ALL vs TRAIN) con TEST
701     # =====
702     # Salidas (VALID):
703     # - <BASE>/validacion/05_KNN_VALIDACION/outputs/knn_test_validation_pairs_all.csv
704     # - <BASE>/validacion/05_KNN_VALIDACION/outputs/knn_test_validation_pairs_train.csv
705     # - <BASE>/validacion/05_KNN_VALIDACION/outputs/knn_test_hit3_by_cluster_cases.csv
706     # - <BASE>/validacion/05_KNN_VALIDACION/outputs/knn_validation_metrics_extended.json
707
708     import os, json, joblib
709     import numpy as np
710     import pandas as pd
711     from datetime import datetime
712     from sklearn.neighbors import NearestNeighbors
713
714     try:
715         from google.colab import drive # type: ignore
716         drive.mount('/content/drive')
717     except Exception:
718         pass
719
720     BASE_DIR = "/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2"
721     ORIGEN_DIR = BASE_DIR
722     DIR_MODELOS = f"{BASE_DIR}/modelos"
723     DIR_VALID = f"{BASE_DIR}/validacion"
724
725     STEP_TAG = "05_KNN_VALIDACION"
726     STEP_DIR = os.path.join(DIR_VALID, STEP_TAG)
727     OUT_DIR = os.path.join(STEP_DIR, "outputs")
728     os.makedirs(OUT_DIR, exist_ok=True)
729
730     # Artefactos kNN
731     nn_train_path = os.path.join(DIR_MODELOS, "knn_pre_train.joblib")
732     nn_all_path = os.path.join(DIR_MODELOS, "knn_pre_all.joblib")
733     knn_params_path = os.path.join(DIR_MODELOS, "knn_params.json")
734     assert os.path.exists(nn_train_path) and os.path.exists(nn_all_path) and os.path.
        exists(knn_params_path), "Faltan artefactos kNN"
735
736     with open(knn_params_path, "r", encoding="utf-8") as f:
737         knn_params = json.load(f)
738     p95_nn1_train = float(knn_params["p95_nn1_train"])
739     INDEX_KEY = knn_params.get("index_key", "knn_row")
740
741     nn_train = joblib.load(nn_train_path)
742     nn_all = joblib.load(nn_all_path)
743
744     # Casos (de Celda E)
745     cases_train_path = os.path.join(DIR_VALID, "04_KNN_PRE", "outputs",
        "knn_train_cases.parquet")
746     cases_all_path = os.path.join(DIR_VALID, "04_KNN_PRE", "outputs",
        "knn_all_cases.parquet")
747     cases_train = pd.read_parquet(cases_train_path)

```

```

748 cases_all = pd.read_parquet(cases_all_path)
749
750 # Datos TEST
751 df_path_candidates = [
752     f"{ORIGEN_DIR}/inputs/df_test_train_v8.xlsx",
753     f"{ORIGEN_DIR}/df_test_train_v8.xlsx",
754 ]
755 df_path = next((p for p in df_path_candidates if os.path.exists(p)), None)
756 assert df_path is not None, "No se encontró df_test_train_v8.xlsx"
757 df = pd.read_excel(df_path)
758 df["Dataset"] = df["Dataset"].astype(str).str.upper()
759 df_test = df[df["Dataset"] == "TEST"].copy().reset_index(drop=True)
760
761 # Columna ID para excluir 'self' por ID
762 id_col = None
763 for cand in ["Identificador", "identificador", "ID", "Id", "id_lote", "id_contrato"]:
764     if cand in df_test.columns: id_col = cand; break
765 if id_col is None:
766     df_test["Identificador"] = [f"test_row_{i}" for i in range(len(df_test))]
767     id_col = "Identificador"
768
769 def sim_percent(d, p95=p95_nn1_train):
770     return max(0.0, 1.0 - (float(d) / p95)) * 100.0 if p95 > 0 else 0.0
771
772 # Consultas (los objetos nn ya están ajustados sobre TRAIN/ALL en NUM-ONLY)
773 dist_all, idx_all = nn_all.kneighbors(n_neighbors=10, return_distance=True, X=None)
774 dist_train, idx_train = nn_train.kneighbors(n_neighbors=10, return_distance=True, X=None)
775
776 # Construcción de pares
777 def build_pairs_csv(df_test, idx_mat, dist_mat, cases_df, out_csv_path,
778     exclude_self_idx=None):
779     rows=[]
780     for i in range(len(df_test)):
781         input_id = df_test.loc[i, id_col]
782         pairs = [(float(dist_mat[i, j]), int(idx_mat[i, j])) for j in range(dist_mat.
783             shape[1])]
784         if exclude_self_idx is not None:
785             pairs = [p for p in pairs if p[1] != exclude_self_idx(i)]
786         if len(pairs) < 2: continue
787         (d1, r1), (d2, r2) = pairs[0], pairs[1]
788         row1 = cases_df.iloc[r1]; row2 = cases_df.iloc[r2]
789         top1_id = row1.get(id_col, row1.get("Identificador", f"row_{r1}"))
790         top2_id = row2.get(id_col, row2.get("Identificador", f"row_{r2}"))
791         rows.append({"input_id": input_id,
792             "top1_id": top1_id, "top1_dist": d1, "top1_sim%": sim_percent(d1
793             ),
794             "top2_id": top2_id, "top2_dist": d2, "top2_sim%": sim_percent(d2
795             )})
796     out = pd.DataFrame(rows)
797     out.to_csv(out_csv_path, index=False)
798     return out
799
800 pairs_all_path = os.path.join(OUT_DIR, "knn_test_validation_pairs_all.csv")
801 pairs_all = build_pairs_csv(df_test, idx_all, dist_all, cases_all, pairs_all_path)
802 print(f"📁 {pairs_all_path}")
803
804 id_to_train_idx={}
805 if id_col in cases_train.columns:
806     for i, r in cases_train.reset_index(drop=True).iterrows():
807         id_to_train_idx[r.get(id_col, f"row_{i}")] = int(r[INDEX_KEY])
808
809 def exclude_self_idx(i):
810     _id = df_test.loc[i, id_col]
811     return id_to_train_idx.get(_id, -99999)
812
813 pairs_train_path = os.path.join(OUT_DIR, "knn_test_validation_pairs_train.csv")
814 pairs_train = build_pairs_csv(df_test, idx_train, dist_train, cases_train,
815     pairs_train_path, exclude_self_idx=exclude_self_idx)
816 print(f"📁 {pairs_train_path}")
817
818 # Métricas rápidas

```

```

814 SIM_THRESHOLD = 60.0
815 def hit_at_least_one(df_pairs, sim_thr=SIM_THRESHOLD):
816     if df_pairs.empty: return 0.0
817     return float((df_pairs[["top1_sim%", "top2_sim%"]].max(axis=1) >= sim_thr).mean()) *
100.0
818
819 metrics = {
820     "built_at": datetime.now().isoformat(),
821     "p95_nnl_train": p95_nnl_train,
822     "hit@2>=60_ALL(%)": hit_at_least_one(pairs_all),
823     "hit@2>=60_TRAIN(%)": hit_at_least_one(pairs_train),
824     "space": "pre_num"
825 }
826 with open(os.path.join(OUT_DIR, "knn_validation_metrics_extended.json"), "w", encoding
="utf-8") as f:
827     json.dump(metrics, f, ensure_ascii=False, indent=2)
828
829 # (Opcional) por cluster si existe
830 if "Cluster_6" in df_test.columns and len(pairs_all):
831     tmp = pairs_all.copy()
832     tmp["Cluster_6"] = df_test["Cluster_6"].values[:len(tmp)]
833     agg = tmp.groupby("Cluster_6").apply(lambda g: float((g[["top1_sim%", "top2_sim%"]
].max(axis=1) >= SIM_THRESHOLD).mean()) * 100.0).reset_index()
834     agg.columns = ["Cluster_6", "hit@2>=60(%)_ALL"]
835     agg.to_csv(os.path.join(OUT_DIR, "knn_test_hit3_by_cluster_cases.csv"), index=
False)
836
837 with open(os.path.join(STEP_DIR, "README_STEP.txt"), "w", encoding="utf-8") as f:
838     f.write(f"{STEP_TAG} generado el {datetime.now().isoformat()} (BASE_DIR={BASE_DIR
}).\n")
839
840 print("☑ Validación KNN completada (todo bajo _v2).")
841
842 """Celda - XGB N ofertantes (desarrollo + validación + guardados producción) desde
df_test_train_v8.xlsx"""
843
844 # =====
845 # XGB N ofertantes - desarrollo + validación + guardados producción
846 # (desde df_test_train_v8.xlsx, sin usar comparativa como fuente)
847 # =====
848 import os, json, joblib
849 import numpy as np
850 import pandas as pd
851 from pathlib import Path
852 from sklearn.model_selection import train_test_split
853 from sklearn.preprocessing import LabelEncoder
854 from sklearn.metrics import accuracy_score, f1_score, recall_score, confusion_matrix
855 import xgboost as xgb
856 import matplotlib.pyplot as plt
857
858 # --- Rutas base (todo bajo _v2) ---
859 BASE_DIR = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
860 DIR_MODELOS = BASE_DIR/"modelos"
861 DIR_VALID = BASE_DIR/"validacion"
862 DIR_COMP = BASE_DIR/"comparativa" # solo para la celda de comparativa (no leemos
aquí)
863 DIR_MODELOS.mkdir(parents=True, exist_ok=True)
864
865 # Validación: subcarpetas (mantengo el nombre que usaste en tu ejemplo)
866 vdir = DIR_VALID / "03_xgb_n_ofertantes"
867 vdir.mkdir(parents=True, exist_ok=True)
868
869 RANDOM_STATE = 42
870
871 # ----- 0) Carga df desde inputs -----
872 df_candidates = [BASE_DIR/"inputs/df_test_train_v8.xlsx", BASE_DIR/
"df_test_train_v8.xlsx"]
873 df_path = next((p for p in df_candidates if p.exists()), None)
874 assert df_path is not None, f"No se encontró df_test_train_v8.xlsx en: {df_candidates
}"
875 df = pd.read_excel(df_path)
876

```

```

877 # ----- 1) Variables preferidas + alias -----
878 XGB_VARS_PREF = [
879     "Presupuesto_licitacion_lote_c", "Plazo_m", "Criterio_precio_p", "N_CPV",
880     "N_clasi_empresa", "Mes_lici", "Intervalo_lici_d_c", "N_lotes",
881     "Tipo_de_Administracion_c", "Tipo_de_procedimiento_c", "C_juicio_valor_p_c",
882     "Provincia2"
883 ]
884 ALIASES = {
885     "Plazo_m": ["Plazo_m_c"],
886     "Criterio_precio_p": ["C_precio_p"],
887 }
888
889 def resolve_vars(preferred, df_cols, aliases):
890     used, mapping, missing = [], {}, []
891     for v in preferred:
892         cands = [v] + aliases.get(v, [])
893         found = next((c for c in cands if c in df_cols), None)
894         if found is None:
895             missing.append(v)
896         else:
897             used.append(found); mapping[v] = found
898     return used, mapping, missing
899
900 # Asegurar Provincia2 existe (derivamos si falta)
901 if "Provincia2" not in df.columns:
902     def _to_cp_str(x):
903         try: return str(int(x)).zfill(5)
904         except Exception:
905             s = str(x); return s if (isinstance(s, str) and s and s.lower()!="nan")
906             else np.nan
907     if "Codigo_Postal_c" in df.columns:
908         df["Provincia2"] = df["Codigo_Postal_c"].apply(_to_cp_str).str[:2].astype(
909             "string")
910     else:
911         df["Provincia2"] = pd.Series([pd.NA]*len(df), dtype="string")
912
913 # Resolver columnas según existan en df
914 XGB_VARS_USED, VAR_MAP, MISSING_PREF = resolve_vars(XGB_VARS_PREF, set(df.columns),
915 ALIASES)
916
917 # Validación de requeridos mínimos para entrenar
918 REQUIRED_MIN = {"Presupuesto_licitacion_lote_c", "N_CPV", "N_clasi_empresa", "Mes_lici",
919 "Intervalo_lici_d_c", "N_lotes", "Tipo_de_Administracion_c",
920 "Tipo_de_procedimiento_c", "C_juicio_valor_p_c"}
921 missing_min = [v for v in REQUIRED_MIN if (v not in VAR_MAP and v not in XGB_VARS_USED
922 )]
923 if "N_ofertantes" not in df.columns:
924     raise ValueError("X 'N_ofertantes' no está en df. No se puede entrenar XGB.")
925 if missing_min:
926     raise ValueError(f"X Faltan columnas mínimas para XGB (tras alias): {missing_min
927 }")
928
929 # ----- 2) Target -> clases 0/1/2 -----
930 def cat_n_ofertantes(n):
931     if pd.isna(n): return np.nan
932     n = int(n)
933     if n <= 4: return 0
934     if n <= 11: return 1
935     return 2
936
937 mask = df["N_ofertantes"].notna()
938 X_raw = df.loc[mask, XGB_VARS_USED].copy()
939 y = df.loc[mask, "N_ofertantes"].apply(cat_n_ofertantes).astype(int)
940 labels_txt = ["BAJO", "MEDIO", "ALTO"]
941
942 # ----- 3) Split 80/20 estratificado -----
943 X_train, X_test, y_train, y_test = train_test_split(
944     X_raw, y, test_size=0.20, random_state=RANDOM_STATE, stratify=y
945 )
946
947 # ----- 4) Encoding (LabelEncoder) en 3 categóricas -----
948 cat_vars_pref = ["Tipo_de_Administracion_c", "Tipo_de_procedimiento_c", "Provincia2"]

```



```

943 # Mapeo preferida->usada para encoders
944 encoders = {}
945 for var_pref in cat_vars_pref:
946     var_used = VAR_MAP.get(var_pref, var_pref)
947     if var_used in X_train.columns:
948         all_cats = pd.concat([X_train[var_used], X_test[var_used]]).astype(str).unique()
949         le = LabelEncoder().fit(all_cats)
950         X_train[var_used] = le.transform(X_train[var_used].astype(str))
951         X_test[var_used] = le.transform(X_test[var_used].astype(str))
952         encoders[var_used] = le # guardamos encoder con el NOMBRE REAL usado
953
954 # Mes_lici numérica
955 mes_col = VAR_MAP.get("Mes_lici", "Mes_lici")
956 if mes_col in X_train.columns and not pd.api.types.is_numeric_dtype(X_train[mes_col]):
957     X_train[mes_col] = pd.to_numeric(X_train[mes_col], errors="coerce").fillna(0).
958     X_test[mes_col] = pd.to_numeric(X_test[mes_col], errors="coerce").fillna(0).
959
960 # ----- 5) Modelo XGB -----
961 xgb_model = xgb.XGBClassifier(
962     n_estimators=50, max_depth=4, learning_rate=0.2,
963     random_state=RANDOM_STATE, eval_metric="mlogloss", verbosity=0
964 )
965 xgb_model.fit(X_train, y_train)
966
967 # ----- 6) Métricas -----
968 y_pred_tr = xgb_model.predict(X_train)
969 y_pred_te = xgb_model.predict(X_test)
970 p_te = xgb_model.predict_proba(X_test)
971
972 acc_tr = accuracy_score(y_train, y_pred_tr)
973 acc_te = accuracy_score(y_test, y_pred_te)
974 f1_per = f1_score(y_test, y_pred_te, average=None)
975 rec_per = recall_score(y_test, y_pred_te, average=None)
976 f1_w = f1_score(y_test, y_pred_te, average="weighted")
977
978 # ----- 7) Matriz de confusión -----
979 cm = confusion_matrix(y_test, y_pred_te, labels=[0,1,2])
980 plt.figure(figsize=(6,5))
981 plt.imshow(cm, interpolation="nearest")
982 plt.title("Matriz de Confusión - XGB N_ofertantes")
983 plt.colorbar()
984 ticks = np.arange(3); plt.xticks(ticks, labels_txt, rotation=45); plt.yticks(ticks,
985 labels_txt)
986 for i in range(cm.shape[0]):
987     for j in range(cm.shape[1]):
988         plt.text(j, i, format(cm[i,j], "d"), ha="center", va="center")
989 plt.ylabel("Reales"); plt.xlabel("Predicciones")
990 plt.tight_layout(); plt.savefig(vdir/"cm_xgb.png", dpi=150); plt.close()
991
992 # ----- 8) Tabla de métricas por clase + total -----
993 tbl = []
994 for i, L in enumerate(labels_txt):
995     casos_reales = int((y_test==i).sum())
996     casos_predichos = int((y_pred_te==i).sum())
997     prec = (cm[i,i]/casos_predichos) if casos_predichos>0 else 0.0
998     tbl.append({"Clase": L, "Casos_Reales": casos_reales, "Casos_Predichos":
999                 casos_predichos,
1000                 "F1_Score": float(f1_per[i]), "Recall": float(rec_per[i]), "Precision":
1001                 float(prec)})
1002 tbl.append({"Clase": "TOTAL", "Casos_Reales": int(len(y_test)), "Casos_Predichos": int(
1003 len(y_test)),
1004             "F1_Score": float(f1_w), "Recall": float(acc_te), "Precision": float(
1005 acc_te)})
1006 pd.DataFrame(tbl).to_csv(vdir/"tabla_metricas_XGB_v8.csv", index=False)
1007
1008 # ----- 9) REAL vs PREDICHO + probas -----
1009 df_pred = pd.DataFrame({
1010     "ID_Test": np.arange(len(y_test)),
1011     "Real": y_test.values,

```



```

1007     "Real_Label": [labels_txt[i] for i in y_test.values],
1008     "Predicho": y_pred_te,
1009     "Predicho_Label": [labels_txt[i] for i in y_pred_te],
1010     "Correcto": (y_test.values == y_pred_te),
1011     "Prob_BAJO": p_te[:,0], "Prob_MEDIO": p_te[:,1], "Prob_ALTO": p_te[:,2],
1012     "Prob_Max": p_te.max(axis=1)
1013 })
1014 df_pred.to_csv(vdir/"real_vs_predicho_XGB_v8.csv", index=False)
1015
1016 # ----- 10) Datasets train/test codificados (traza) -----
1017 df_train_final = pd.DataFrame(X_train, columns=X_train.columns)
1018 df_train_final["N_ofertantes_cat"] = y_train.values
1019 df_train_final["N_ofertantes_label"] = ["BAJO" if i==0 else ("MEDIO" if i==1 else
1020 "ALTO") for i in y_train.values]
1021 df_test_final = pd.DataFrame(X_test, columns=X_test.columns)
1022 df_test_final["N_ofertantes_cat"] = y_test.values
1023 df_test_final["N_ofertantes_label"] = ["BAJO" if i==0 else ("MEDIO" if i==1 else
1024 "ALTO") for i in y_test.values]
1025 df_train_final.to_csv(vdir/"df_train_XGB_v8.csv", index=False)
1026 df_test_final.to_csv(vdir/"df_test_XGB_v8.csv", index=False)
1027
1028 # ----- 11) Importancias -----
1029 imp = getattnr(xgb_model, "feature_importances ", None)
1030 if imp is not None and len(imp) == df_train_final.shape[1]-2:
1031     pd.DataFrame({"Variable": df_train_final.columns[:-2], "Importancia": imp})\
1032         .sort_values("Importancia", ascending=False)\
1033         .to_csv(vdir/"importancia_variables_XGB_v8.csv", index=False)
1034
1035 # ----- 12) Pack para PRODUCCIÓN -----
1036 pack = {
1037     "modelo": xgb_model,
1038     "encoders": encoders, # dict: {col_usada: LabelEncoder}
1039     "variables_predictoras": XGB_VARS_USED, # NOMBRES REALES usados (tras alias)
1040     "labels": ["BAJO", "MEDIO", "ALTO"],
1041     "alias_mapping": VAR_MAP, # mapping preferida->usada (traza)
1042     "metricas_test": {
1043         "accuracy": float(acc_te),
1044         "f1_scores": [float(x) for x in f1_per],
1045         "recall_scores": [float(x) for x in rec_per],
1046         "f1_weighted": float(f1_w)
1047     },
1048     "version": "v8",
1049     "random_state": int(RANDOM_STATE)
1050 }
1051 joblib.dump(pack, DIR_MODELOS / "model_Noferta_XGB_v8.pkl")
1052
1053 # ----- 13) Predict de ejemplo -----
1054 ejemplo = X_test.iloc[:1].copy()
1055 ejemplo.to_csv(vdir/"predict_XGB_v8.csv", index=False)
1056
1057 # ----- 14) Resumen + explicación -----
1058 with open(vdir/"metrics.json", "w", encoding="utf-8") as f:
1059     json.dump({"accuracy_train": float(acc_tr), "accuracy_test": float(acc_te),
1060         "f1_weighted": float(f1_w),
1061         "f1_per_class": dict(zip(["BAJO", "MEDIO", "ALTO"], [float(x) for x in
1062         f1_per])),
1063         "recall_per_class": dict(zip(["BAJO", "MEDIO", "ALTO"], [float(x) for x
1064         in rec_per]))},
1065         f, indent=2, ensure_ascii=False)
1066
1067 with open(vdir/"explicacion_tecnica.txt", "w", encoding="utf-8") as f:
1068     f.write(
1069         "XGB multiclase (BAJO/MEDIO/ALTO) para N_ofertantes con split 80/20 (rs=42). "
1070         "Resolución de alias (p.ej., Criterio_precio_p/C_precio_p; "
1071         "Plazo_m/Plazo_m_c). "
1072         "Encoding: LabelEncoder en Tipo_de_Administracion_c, Tipo_de_procedimiento_c, "
1073         "Provincia2 (usando nombres reales tras alias). "
1074         "Mes_licí numérica. Pack de producción guarda variables usadas y el mapping "
1075         "de alias."
1076     )
1077
1078 print("✅ XGB N_ofertantes entrenado y guardado")

```

```

1072 print("→ modelos/: model Noferta_XGB_v8.pkl")
1073 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")

1074
1075 """Celda C – META del XGB (baja_comp): políticas y fallbacks"""
1076
1077 # =====
1078 # META – XGB baja_comp (políticas de inputs)
1079 # =====
1080 import json, joblib, numpy as np, pandas as pd
1081 from pathlib import Path
1082
1083 BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
1084 DIR_MODELOS = BASE / "modelos"
1085 DIR_INPUTS = BASE / "inputs"
1086
1087 pack_path = DIR_MODELOS / "model_Noferta_XGB_v8.pkl"
1088 assert pack_path.exists(), "Falta modelos/model_Noferta_XGB_v8.pkl"
1089 pack = joblib.load(pack_path)
1090
1091 # Cargar df de partida para medir modas (preferencia inputs/df_test_train_v8.xlsx)
1092 df_path = None
1093 for p in [DIR_INPUTS/"df_test_train_v8.xlsx", BASE/"df_test_train_v8.xlsx"]:
1094     if p.exists(): df_path = p; break
1095 assert df_path is not None, "No encuentro df_test_train_v8.xlsx para extraer modas de
categóricas."
1096 df_all = pd.read_excel(df_path)
1097
1098 # Columnas y encoders reales del pack
1099 xgb_vars = list(pack.get("variables_predictoras", []))
1100 xgb_encs = pack.get("encoders", {}) # {col_usada: LabelEncoder}
1101
1102 # Mapa fallback por columna categórica = moda en df_all (si no está, usamos primer
class_del encoder)
1103 fallback_class_por_col = {}
1104 for col, le in xgb_encs.items():
1105     if col in df_all.columns:
1106         s = df_all[col].astype(str)
1107         if len(s.dropna()) > 0:
1108             moda = s.value_counts(dropna=True).idxmax()
1109             # Si la moda no está en el encoder (variantes), elegimos la clase más
frecuente del encoder
1110             if moda not in set(map(str, le.classes_)):
1111                 # fallback a la clase con más ocurrencias en df_all entre las classes_
1112                 vc = {cls: int((s == str(cls)).sum()) for cls in le.classes_}
1113                 moda = max(vc, key=vc.get)
1114                 fallback_class_por_col[col] = str(moda)
1115             else:
1116                 fallback_class_por_col[col] = str(le.classes_[0])
1117         else:
1118             fallback_class_por_col[col] = str(le.classes_[0])
1119
1120 xgb_meta = {
1121     "artifact": "model_Noferta_XGB_v8.pkl",
1122     "variables_predictoras": xgb_vars,
1123     "encoders_columns": list(xgb_encs.keys()),
1124     "labels": list(pack.get("labels", [])),
1125     "policies": {
1126         "missing_numeric_policy": "fill_0",
1127         "cat_unknown_policy": "fallback_to_mode", # usar moda por columna
1128         "fallback_class_por_col": fallback_class_por_col,
1129         "threshold_bajo": 0.60
1130     },
1131     "version": pack.get("version", "v8"),
1132     "random_state": int(pack.get("random_state", 42))
1133 }
1134 (DIR_MODELOS / "xgb_meta.json").write_text(json.dumps(xgb_meta, indent=2, ensure_ascii
=False), encoding="utf-8")
1135 print("☑ Guardado modelos/xgb_meta.json")
1136
1137 """RF base (PRE mínimas) – desarrollo + validación + guardados"""

```

```

1138
1139 # =====
1140 # RF base (PRE mínimas) – desarrollo + validación + guardados
1141 # =====
1142 import os, json, joblib, numpy as np, pandas as pd, matplotlib.pyplot as plt
1143 from pathlib import Path
1144 from sklearn.compose import ColumnTransformer
1145 from sklearn.pipeline import Pipeline
1146 from sklearn.preprocessing import OneHotEncoder, StandardScaler
1147 from sklearn.impute import SimpleImputer
1148 from sklearn.ensemble import RandomForestClassifier
1149 from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
1150 from sklearn.metrics import accuracy_score, f1_score, confusion_matrix
1151
1152 # --- Rutas base ---
1153 BASE_DIR = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
1154 DIR_MODELOS = BASE_DIR / "modelos"
1155 DIR_VALID = BASE_DIR / "validacion"
1156 DIR_MODELOS.mkdir(parents=True, exist_ok=True)
1157
1158 vdir = DIR_VALID / "04_rf_base"
1159 vdir.mkdir(parents=True, exist_ok=True)
1160
1161 RANDOM_STATE = 42
1162
1163 # --- Cargar df y splits TRAIN/TEST ---
1164 df_candidates = [BASE_DIR/"inputs/df_test_train_v8.xlsx", BASE_DIR/
1165 "df_test_train_v8.xlsx"]
1166 df_path = next((p for p in df_candidates if p.exists()), None)
1167 assert df_path is not None, f"No se encontró df_test_train_v8.xlsx en: {df_candidates}"
1168 df = pd.read_excel(df_path)
1169
1170 assert "Cluster_6" in df.columns, "X Falta 'Cluster_6'."
1171 assert "Dataset" in df.columns, "X Falta 'Dataset' (TRAIN/TEST)."
```

```

1172 df["Cluster_6"] = df["Cluster_6"].astype(str)
1173 df["Dataset"] = df["Dataset"].astype(str).str.upper()
1174
1175 df_train = df[df["Dataset"] == "TRAIN"].copy()
1176 df_test = df[df["Dataset"] == "TEST"].copy()
1177
1178 # 1) Variables PRE mínimas acordadas (sin Provincia2) + alias
1179 MIN_PRE_PREF = [
1180     "Presupuesto_licitacion_lote_c", "Plazo_m", "Criterio_precio_p", "C_precio_p",
1181     "N_CPV", "N_clasi_empresa", "Mes_lici", "Intervalo_lici_d_c", "N_lotes",
1182     "Tipo_de_Administracion_c", "Tipo_de_procedimiento_c", "C_juicio_valor_p_c"
1183 ]
1184 ALIASES = {
1185     "Criterio_precio_p": ["C_precio_p"], # preferimos Criterio_precio_p; si no,
1186     "C_precio_p": ["C_precio_p"],
1187     "Plazo_m": ["Plazo_m_c"], # preferimos Plazo_m; si no, Plazo_m_c
1188 }
1189
1190 def resolve_vars(preferred, df_cols, aliases):
1191     used = []
1192     mapping = {}
1193     # Regla especial para la variable de precio: elegir UNA (preferida si existe)
1194     precio_chosen = None
1195     for v in preferred:
1196         cands = [v] + aliases.get(v, [])
1197         found = next((c for c in cands if c in df_cols), None)
1198         if found is None:
1199             continue
1200         if v in ["Criterio_precio_p", "C_precio_p"]:
1201             if precio_chosen is not None:
1202                 # ya elegimos una de las dos, saltamos la otra
1203                 continue
1204             # si existe Criterio_precio_p, tomamos esa y marcamos; si no, tomamos
1205             # C_precio_p
1206             if "Criterio_precio_p" in df_cols:
1207                 precio_chosen = "Criterio_precio_p"
1208             else:
1209                 precio_chosen = "C_precio_p"
1210         used.append(precio_chosen)

```

```

1206         mapping["Criterio_precio_p"] = "Criterio_precio_p"
1207     elif "C_precio_p" in df_cols:
1208         precio_chosen = "C_precio_p"
1209         used.append("C_precio_p")
1210         mapping["Criterio_precio_p"] = "C_precio_p" # mapping a la usada real
1211     continue
1212 # Plazo_m vs Plazo_m_c: elegimos una
1213 if v == "Plazo_m":
1214     if "Plazo_m" in df_cols:
1215         used.append("Plazo_m"); mapping["Plazo_m"] = "Plazo_m"
1216     elif "Plazo_m_c" in df_cols:
1217         used.append("Plazo_m_c"); mapping["Plazo_m"] = "Plazo_m_c"
1218     continue
1219     used.append(found); mapping[v] = found
1220 # quitar duplicados conservando orden
1221 seen=set(); used_uniq=[]
1222 for c in used:
1223     if c not in seen:
1224         seen.add(c); used_uniq.append(c)
1225 return used_uniq, mapping
1226
1227 MIN_PRE_USED, VAR_MAP = resolve_vars(MIN_PRE_PREF, set(df.columns), ALIASES)
1228 assert len(MIN_PRE_USED) > 0, "X No se resolvieron variables PRE mínimas."
1229
1230 # 2) Preparar X/y (TRAIN/TEST) – asegurar Mes_lici categórica para RF
1231 Xtr = df_train[MIN_PRE_USED].copy()
1232 Xte = df_test[MIN_PRE_USED].copy()
1233 ytr = df_train["Cluster_6"].astype(str).copy()
1234 yte = df_test["Cluster_6"].astype(str).copy()
1235
1236 mes_col = VAR_MAP.get("Mes_lici", "Mes_lici")
1237 if mes_col in Xtr.columns:
1238     Xtr[mes_col] = Xtr[mes_col].astype("Int64").astype("category")
1239     Xte[mes_col] = Xte[mes_col].astype("Int64").astype("category")
1240
1241 # 3) Preprocesamiento + modelo
1242 try:
1243     oh = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
1244 except TypeError:
1245     oh = OneHotEncoder(handle_unknown="ignore", sparse=False)
1246
1247 num_cols = Xtr.select_dtypes(include=[np.number]).columns.tolist()
1248 cat_cols = [c for c in Xtr.columns if c not in num_cols]
1249
1250 prep = ColumnTransformer([
1251     ("num", Pipeline([("imp", SimpleImputer(strategy="median")), ("sc", StandardScaler())]), num_cols),
1252     ("cat", Pipeline([("imp", SimpleImputer(strategy="most_frequent")), ("oh", oh)]), cat_cols),
1253 ])
1254
1255 rf = RandomForestClassifier(random_state=RANDOM_STATE, n_jobs=-1)
1256 pipe = Pipeline([("prep", prep), ("clf", rf)])
1257
1258 # 4) Tuning ligero
1259 param_grid = {
1260     "clf__n_estimators": [600, 900, 1100],
1261     "clf__max_depth": [None, 18, 24],
1262     "clf__min_samples_split": [2, 5, 10],
1263     "clf__min_samples_leaf": [1, 2, 4],
1264     "clf__max_features": ["sqrt", 0.5],
1265     "clf__class_weight": [None, "balanced_subsample"]
1266 }
1267 cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_STATE)
1268 rs = RandomizedSearchCV(
1269     pipe, param_distributions=param_grid, n_iter=28,
1270     cv=cv, scoring="f1_macro", random_state=RANDOM_STATE, n_jobs=-1, verbose=0
1271 )
1272 rs.fit(Xtr, ytr)
1273 rf_base = rs.best_estimator_
1274
1275 # 5) Predicciones y métricas

```

```

1276 pred = rf_base.predict(Xte)
1277 proba = rf_base.predict_proba(Xte)
1278 acc = accuracy_score(yte, pred)
1279 flm = f1_score(yte, pred, average="macro")
1280
1281 def per_class_acc(y_true, y_pred):
1282     out={}
1283     for cl in sorted(pd.unique(y_true), key=lambda s:int(s)):
1284         m = (y_true==cl)
1285         out[cl] = float((y_pred[m]==y_true[m]).mean()) if m.any() else np.nan
1286     return out
1287
1288 acc_c = per_class_acc(yte, pred)
1289
1290 # 6) Guardados
1291 joblib.dump(rf_base, DIR_MODELOS / "rf_base.pkl")
1292 pd.DataFrame({"y_true": yte, "y_pred": pred}).to_csv(vdir / "pred_test_rf_base.csv",
1293 index=False)
1294 pd.Series(acc_c).to_csv(vdir / "accuracy_por_clase.csv", header=["accuracy"])
1295 with open(vdir / "summary.json", "w", encoding="utf-8") as f:
1296     json.dump({
1297         "accuracy": float(acc),
1298         "macro_f1": float(flm),
1299         "features_used": MIN_PRE_USED,
1300         "alias_mapping": VAR_MAP,
1301         "best_params": rs.best_params_
1302     }, f, indent=2, ensure_ascii=False)
1303
1304 # Matriz de confusión
1305 labs = sorted(pd.unique(pd.concat([ytr, yte])), key=lambda s:int(s))
1306 cm = confusion_matrix(yte, pred, labels=labs)
1307 plt.figure(figsize=(6,5))
1308 plt.imshow(cm, interpolation="nearest")
1309 plt.title("Matriz de Confusión - RF base")
1310 plt.colorbar()
1311 ticks=np.arange(len(labs)); plt.xticks(ticks, labs, rotation=45, ha="right"); plt.yticks(
1312 ticks, labs)
1313 for i in range(cm.shape[0]):
1314     for j in range(cm.shape[1]):
1315         plt.text(j,i,format(cm[i,j], 'd'), ha="center", va="center")
1316 plt.ylabel("Reales"); plt.xlabel("Predicciones")
1317 plt.tight_layout(); plt.savefig(vdir / "cm_rf_base.png", dpi=150); plt.close()
1318
1319 with open(vdir / "explicacion_tecnica.txt", "w", encoding="utf-8") as f:
1320     f.write(
1321         "RF base entrenado SOLO con variables PRE mínimas (sin Provincia2). "
1322         "Pipeline: imputación (mediana/moda), OHE(handle_unknown='ignore'), escalado
1323         numérico (var≈1). "
1324         "Tuning ligero con RandomizedSearchCV (f1_macro, rs=42). Artefacto final en
1325         /modelos/rf_base.pkl."
1326     )
1327
1328 print("✅ RF base entrenado y guardado en modelos/rf_base.pkl")
1329 print(f"[TEST] Acc={acc:.4f} | Macro-F1={flm:.4f} | Vars usadas={len(MIN_PRE_USED)}")
1330
1331 """Celda D - META del RF base (features crudas exactas + dtypes esperados)"""
1332
1333 # =====
1334 # META - RF base (contrato de features)
1335 # =====
1336 import json, joblib, sklearn
1337 from pathlib import Path
1338
1339 BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
1340 DIR_MODELOS = BASE / "modelos"
1341
1342 rf_path = DIR_MODELOS / "rf_base.pkl"
1343 assert rf_path.exists(), "Falta modelos/rf_base.pkl"
1344 rf_base = joblib.load(rf_path)
1345
1346 # Extraer columnas crudas esperadas por el ColumnTransformer
1347 def _cols_from_pipe(pipe):

```

```

1344     try:
1345         trfs = pipe.named_steps["prep"].transformers_
1346         cols = {}
1347         for name, _, c in pipe.named_steps["prep"].transformers_:
1348             cols[name] = list(c) if isinstance(c, list) else []
1349         return cols
1350     except Exception:
1351         return {"num": [], "cat": []}
1352
1353 cols = _cols_from_pipe(rf_base)
1354 alias_mapping = globals().get("VAR_MAP", {}) # si existe en el notebook, lo
aprovechamos
1355
1356 rf_meta = {
1357     "artifact": "rf_base.pkl",
1358     "expected_raw_features": cols,
1359     "alias_mapping_used_in_train": alias_mapping,
1360     "dtypes_expect": {"Mes_lici": "category"},
1361     "sklearn_version": sklearn.__version__
1362 }
1363 (DIR_MODELOS / "rf_base_meta.json").write_text(json.dumps(rf_meta, indent=2,
ensure_ascii=False), encoding="utf-8")
1364 print("✓ Guardado modelos/rf_base_meta.json")
1365
1366 """RF_124_BASE+BC (train) + Ensemble "seguro" BASE+BC (test)"""
1367
1368 # =====
1369 # Celda única - RF_124_BASE+BC (train) + Ensemble "seguro" BASE+BC (test)
1370 # - Entrena especialista 1/2/4 con MIN_PRE + baja_comp (XGB) duplicada (k=5)
1371 # - Construye ensemble seguro vs RF base y genera la gráfica final
1372 # - Guarda artefactos de PRODUCCIÓN (rf_124_basebc.pkl + ensemble_meta.json)
1373 # - Guarda validaciones (CSV/PNG/JSON)
1374 # =====
1375 import os, json, joblib, numpy as np, pandas as pd, matplotlib.pyplot as plt
1376 from pathlib import Path
1377 from sklearn.compose import ColumnTransformer
1378 from sklearn.pipeline import Pipeline
1379 from sklearn.preprocessing import OneHotEncoder, StandardScaler
1380 from sklearn.impute import SimpleImputer
1381 from sklearn.ensemble import RandomForestClassifier
1382 from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
1383 from sklearn.metrics import accuracy_score, f1_score, confusion_matrix
1384
1385 # ----- Configuración -----
1386 BASE_DIR = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
1387 DIR_MODELOS = BASE_DIR / "modelos"
1388 DIR_VALID = BASE_DIR / "validacion"
1389 DIR_MODELOS.mkdir(parents=True, exist_ok=True)
1390 (DIR_VALID / "05_rf_124_basebc").mkdir(parents=True, exist_ok=True)
1391 (DIR_VALID / "06_ensemble_basebc").mkdir(parents=True, exist_ok=True)
1392
1393 RANDOM_STATE = 42
1394 CLASSES_124 = {"1", "2", "4"}
1395 BAJA_COMP_DUP = 5 # ← regla acordada (k=5)
1396 TH_BAJO = 0.55
1397 ALLOWED_DROP_1 = 0.05 # ← máx 5 pp
1398 REQUIRE_IMPROVE_4 = True
1399 REQUIRE_IMPROVE_GLOBAL = True
1400 MARGIN_MIN = 0.03
1401
1402 vdir_model = DIR_VALID / "05_rf_124_basebc"
1403 vdir_ens = DIR_VALID / "06_ensemble_basebc"
1404
1405 # ----- 0) Cargar datos y prerequisites -----
1406 df_candidates = [BASE_DIR/"inputs/df_test_train_v8.xlsx", BASE_DIR/
"df_test_train_v8.xlsx"]
1407 df_path = next((p for p in df_candidates if p.exists()), None)
1408 assert df_path is not None, f"No se encontró df_test_train_v8.xlsx en: {df_candidates
}"
1409 df = pd.read_excel(df_path)
1410
1411 assert "Cluster_6" in df.columns, "X Falta 'Cluster_6'."

```



```

1412 assert "Dataset" in df.columns, "X Falta 'Dataset' (TRAIN/TEST)."
1413 df["Cluster_6"] = df["Cluster_6"].astype(str)
1414 df["Dataset"] = df["Dataset"].astype(str).str.upper()
1415 df_train = df[df["Dataset"] == "TRAIN"].copy()
1416 df_test = df[df["Dataset"] == "TEST"].copy()
1417
1418 rf_base_path = DIR_MODELOS / "rf_base.pkl"
1419 xgb_pack_path = DIR_MODELOS / "model_Noferta_XGB_v8.pkl"
1420 assert rf_base_path.exists(), f"X Falta {rf_base_path}. Entrena antes RF base."
1421 assert xgb_pack_path.exists(), f"X Falta {xgb_pack_path}. Entrena antes XGB
N_ofertantes."
1422 rf_base = joblib.load(rf_base_path)
1423 xgb_pack = joblib.load(xgb_pack_path)
1424
1425 # ----- 1) Utilidades -----
1426 MIN_PRE_PREF = [
1427     "Presupuesto_licitacion_lote_c", "Plazo_m", "Criterio_precio_p", "C_precio_p",
1428     "N_CPV", "N_clasi_empresa", "Mes_lici", "Intervalo_lici_d_c", "N_lotes",
1429     "Tipo_de_Administracion_c", "Tipo_de_procedimiento_c", "C_juicio_valor_p_c"
1430 ]
1431 ALIASES = {"Criterio_precio_p": ["C_precio_p"], "Plazo_m": ["Plazo_m_c"]}
1432
1433 def resolve_vars(preferred, df_cols, aliases):
1434     used, mapping = [], {}
1435     precio_chosen = None
1436     for v in preferred:
1437         cand = [v] + aliases.get(v, [])
1438         found = next((c for c in cand if c in df_cols), None)
1439         if found is None:
1440             continue
1441         if v in ["Criterio_precio_p", "C_precio_p"]:
1442             if precio_chosen is not None:
1443                 continue
1444             if "Criterio_precio_p" in df_cols:
1445                 precio_chosen = "Criterio_precio_p"
1446                 used.append("Criterio_precio_p"); mapping["Criterio_precio_p"] =
"Criterio_precio_p"
1447             elif "C_precio_p" in df_cols:
1448                 precio_chosen = "C_precio_p"
1449                 used.append("C_precio_p"); mapping["Criterio_precio_p"] = "C_precio_p"
1450             continue
1451         if v == "Plazo_m":
1452             if "Plazo_m" in df_cols:
1453                 used.append("Plazo_m"); mapping["Plazo_m"] = "Plazo_m"; continue
1454             if "Plazo_m_c" in df_cols:
1455                 used.append("Plazo_m_c"); mapping["Plazo_m"] = "Plazo_m_c"; continue
1456         used.append(found); mapping[v] = found
1457     # único conservando orden
1458     seen = set(); used_uniq = []
1459     for c in used:
1460         if c not in seen:
1461             seen.add(c); used_uniq.append(c)
1462     return used_uniq, mapping
1463
1464 MIN_PRE_USED, VAR_MAP = resolve_vars(MIN_PRE_PREF, set(df_train.columns), ALIASES)
1465
1466 # Provincia2 para XGB si falta
1467 def _to_cp_str(x):
1468     try: return str(int(x)).zfill(5)
1469     except Exception:
1470         s = str(x); return s if (isinstance(s, str) and s and s.lower() != "nan") else np
.nan
1471
1472 if "Provincia2" not in df.columns:
1473     if "Codigo_Postal_c" in df.columns:
1474         df["Provincia2"] = df["Codigo_Postal_c"].apply(_to_cp_str).str[:2].astype(
"string")
1475     else:
1476         df["Provincia2"] = pd.Series([pd.NA]*len(df), dtype="string")
1477 # (recalcular train/test por si añadimos la columna)
1478 df_train = df[df["Dataset"] == "TRAIN"].copy()
1479 df_test = df[df["Dataset"] == "TEST"].copy()
1480

```



```

1480 # Pack XGB → predictor de baja_comp (BAJO/NO BAJO con umbral 0.55)
1481 xgb_model = xgb_pack["modelo"]
1482 xgb_vars = xgb_pack["variables_predictoras"] # nombres REALES usados en el pack
1483 xgb_encs = xgb_pack["encoders"] # {col_usada: LabelEncoder}
1484 xgb_labels = list(xgb_pack["labels"])
1485 idx_bajo = xgb_labels.index("BAJO") if "BAJO" in xgb_labels else 0
1486
1487 def comp_predict_bajo(df_in: pd.DataFrame) -> pd.Series:
1488     Xp = pd.DataFrame(index=df_in.index)
1489     for col in xgb_vars:
1490         Xp[col] = df_in[col] if col in df_in.columns else 0
1491     for col, le in xgb_encs.items():
1492         if col in Xp.columns:
1493             Xp[col] = le.transform(Xp[col].astype(str))
1494         else:
1495             Xp[col] = 0
1496     if "Mes_lici" in Xp.columns and not pd.api.types.is_numeric_dtype(Xp["Mes_lici"]):
1497         Xp["Mes_lici"] = pd.to_numeric(Xp["Mes_lici"], errors="coerce").fillna(0).
1498         astype(int)
1499     p = xgb_model.predict_proba(Xp[xgb_vars])[:, idx_bajo]
1500     return pd.Series((p >= TH_BAJO).astype(np.int8), index=df_in.index, name=
1501         "baja_comp")
1502
1503 # Dataset BASE+BC (MIN_PRE + baja_comp categórica duplicada k veces)
1504 def build_basebc(df_in: pd.DataFrame) -> pd.DataFrame:
1505     X = df_in[MIN_PRE_USED].copy()
1506     mes_col = VAR_MAP.get("Mes_lici", "Mes_lici")
1507     if mes_col in X.columns:
1508         X[mes_col] = X[mes_col].astype("Int64").astype("category")
1509     bc = comp_predict_bajo(df_in).astype("category")
1510     X["baja_comp"] = bc
1511     for k in range(1, BAJA_COMP_DUP+1):
1512         X[f"baja_comp_dup{k}"] = X["baja_comp"]
1513     return X
1514
1515 # ----- 2) Preparar datos especialista 1/2/4 -----
1516 Xtr_basebc = build_basebc(df_train)
1517 Xte_basebc = build_basebc(df_test)
1518 pd.concat([df_train[["Cluster_6"]], Xtr_basebc], axis=1).to_csv(vdir_model/
1519     "train_BASEBC_124.csv", index=False)
1520 pd.concat([df_test[["Cluster_6"]], Xte_basebc], axis=1).to_csv(vdir_model/
1521     "test_BASEBC_124.csv", index=False)
1522
1523 mask_tr = df_train["Cluster_6"].isin(CLASSES_124)
1524 X124_tr = Xtr_basebc.loc[mask_tr].copy()
1525 y124_tr = df_train.loc[mask_tr, "Cluster_6"].astype(str).copy()
1526
1527 # ----- 3) Entrenar RF_124_BASE+BC -----
1528 num_cols = X124_tr.select_dtypes(include=[np.number]).columns.tolist()
1529 cat_cols = [c for c in X124_tr.columns if c not in num_cols]
1530 try:
1531     oh = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
1532 except TypeError:
1533     oh = OneHotEncoder(handle_unknown="ignore", sparse=False)
1534
1535 prep = ColumnTransformer([
1536     ("num", Pipeline([("imp", SimpleImputer(strategy="median")), ("sc", StandardScaler
1537         ())]), num_cols),
1538     ("cat", Pipeline([("imp", SimpleImputer(strategy="most_frequent")), ("oh", oh)]),
1539         cat_cols),
1540 ])
1541 rf = RandomForestClassifier(random_state=RANDOM_STATE, n_jobs=-1)
1542 pipe = Pipeline([("prep", prep), ("clf", rf)])
1543
1544 param_grid = {
1545     "clf__n_estimators": [600, 900, 1100],
1546     "clf__max_depth": [None, 18, 24],
1547     "clf__min_samples_split": [2, 5, 10],
1548     "clf__min_samples_leaf": [1, 2, 4],
1549     "clf__max_features": ["sqrt", 0.5],
1550     "clf__class_weight": [None, "balanced_subsample"]
1551 }

```

```

1546 cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_STATE)
1547 rs = RandomizedSearchCV(pipe, param_distributions=param_grid, n_iter=28, cv=cv,
1548                          scoring="f1_macro", random_state=RANDOM_STATE, n_jobs=-1,
                          verbose=0)
1549 rs.fit(X124_tr, y124_tr)
1550 rf_124_basebc = rs.best_estimator_
1551 joblib.dump(rf_124_basebc, DIR_MODELOS / "rf_124_basebc.pkl")
1552 print("📁 Guardado: modelos/rf_124_basebc.pkl")
1553
1554 # ----- 4) Predicciones base y especialista en TEST -----
1555 # A) RF base
1556 def required_cols_from_pipe(pipe):
1557     try:
1558         trfs = pipe.named_steps["prep"].transformers_
1559         cols = []
1560         for _, cols_i in trfs:
1561             if isinstance(cols_i, list): cols.extend(cols_i)
1562             # único preservando orden
1563         out = []
1564         for c in cols:
1565             if c not in out: out.append(c)
1566         return out
1567     except Exception:
1568         return None
1569
1570 req_base_cols = required_cols_from_pipe(rf_base) or [c for c in df_test.columns if c
1571 != "Cluster_6"]
1572 Xte_base = df_test[[c for c in req_base_cols if c in df_test.columns]].copy()
1573 if "Mes_lici" in Xte_base.columns:
1574     Xte_base["Mes_lici"] = Xte_base["Mes_lici"].astype("Int64").astype("category")
1575
1576 proba_base = rf_base.predict_proba(Xte_base)
1577 labels_base = [str(c) for c in rf_base.named_steps["clf"].classes_]
1578 pred_base = np.array([labels_base[i] for i in np.argmax(proba_base, axis=1)])
1579
1580 # B) Especialista BASE+BC
1581 proba_bb = rf_124_basebc.predict_proba(Xte_basebc)
1582 labels_bb = [str(c) for c in rf_124_basebc.named_steps["clf"].classes_]
1583 pred_bb = np.array([labels_bb[i] for i in np.argmax(proba_bb, axis=1)])
1584
1585 y_true = df_test["Cluster_6"].astype(str).to_numpy()
1586
1587 # ----- 5) Ensemble "seguro" (flips iterativos con margen)
1588 -----
1589 def safe_ensemble(pred_b, proba_b, labels_b, pred_124, proba_124, labels_124):
1590     pred = pred_b.copy()
1591     scope = np.isin(pred_b, list(CLASSES_124))
1592     idx_b = {lb:i for i, lb in enumerate(labels_b)}
1593     idx_s = {lb:i for i, lb in enumerate(labels_124)}
1594     cand_s = []
1595     for i in np.where(scope)[0]:
1596         base_cls = pred_b[i]
1597         alt_cls = pred_124[i]
1598         if alt_cls != base_cls and alt_cls in CLASSES_124 and base_cls in idx_b and
1599 alt_cls in idx_s:
1600             mb = proba_b[i][idx_b[base_cls]]
1601             ma = proba_124[i][idx_s[alt_cls]]
1602             if (ma - mb) >= MARGIN_MIN:
1603                 cand_s.append((ma-mb, i, alt_cls))
1604     cand_s.sort(reverse=True)
1605
1606 def acc_cls(y, yp, cls):
1607     m=(y==cls);
1608     return (yp[m]==y[m]).mean() if m.any() else np.nan
1609
1610 acc0 = accuracy_score(y_true, pred_b)
1611 acc1_0 = acc_cls(y_true, pred_b, "1")
1612 acc4_0 = acc_cls(y_true, pred_b, "4")
1613
1614 flipped = []
1615 cur = pred.copy()
1616 for _, i, alt in cand_s:

```

```

1614         tmp = cur.copy()
1615         tmp[i] = alt
1616         acc = accuracy_score(y_true, tmp)
1617         acc1 = acc_cls(y_true, tmp, "1")
1618         acc4 = acc_cls(y_true, tmp, "4")
1619         condG = (not REQUIRE_IMPROVE_GLOBAL) or (acc >= acc0 - 1e-12)
1620         cond4 = (not REQUIRE_IMPROVE_4) or (acc4 >= acc4_0 - 1e-12)
1621         cond1 = (np.isnan(acc1_0) or np.isnan(acc1)) or (acc1 >= acc1_0 -
1622             ALLOWED_DROP_1 - 1e-12)
1623         if condG and cond4 and cond1:
1624             cur = tmp
1625             acc0, acc1_0, acc4_0 = acc, acc1, acc4
1626             flipped.append(i)
1627     return cur, flipped
1628
1629 ens_bb, flips_bb = safe_ensemble(pred_base, proba_base, labels_base, pred_bb, proba_bb,
1630     , labels_bb)
1631
1632 # ----- 6) Métricas + gráfica final -----
1633 def per_class_acc(y, yp):
1634     d={}
1635     for lb in sorted(pd.unique(y), key=lambda s:int(s)):
1636         m=(y==lb); d[str(lb)] = (yp[m]==y[m]).mean() if m.any() else np.nan
1637     return pd.Series(d)
1638
1639 acc_b = accuracy_score(y_true, pred_base); f1_b = f1_score(y_true, pred_base,
1640     average="macro")
1641 acc_bb = accuracy_score(y_true, ens_bb); f1_bb = f1_score(y_true, ens_bb,
1642     average="macro")
1643
1644 pa_b = per_class_acc(y_true, pred_base)
1645 pa_bb = per_class_acc(y_true, ens_bb)
1646
1647 print(f"[TEST] RF base -> Acc={acc_b:.4f} | Macro-F1={f1_b:.4f}")
1648 print(f"[TEST] Ensemble BASE+BC -> Acc={acc_bb:.4f} | Macro-F1={f1_bb:.4f} | flips={
1649     len(flips_bb)}")
1650
1651 # Gráfica final (barras por clase: base vs ensemble BASE+BC)
1652 labs = sorted(pa_b.index, key=lambda s:int(s))
1653 x = np.arange(len(labs)); w = 0.35
1654 plt.figure(figsize=(11,4))
1655 plt.bar(x - w/2, [pa_b[l] for l in labs], width=w, label="RF base")
1656 plt.bar(x + w/2, [pa_bb[l] for l in labs], width=w, label="Ensemble BASE+BC")
1657 for i,l in enumerate(labs):
1658     for off,val in [(-w/2,pa_b[l]),(w/2,pa_bb[l])]:
1659         plt.text(x[i]+off, val+0.01, f"{val:.2f}", ha="center", va="bottom", fontsize=
1660             9)
1661 plt.xticks(x, labs); plt.ylabel("Accuracy"); plt.title("Accuracy por clase - RF base
1662     vs Ensemble BASE+BC")
1663 plt.legend(); plt.tight_layout(); plt.savefig(vdir_ens /
1664     "acc_por_clase_base_vs_ensemble.png", dpi=160); plt.close()
1665
1666 # Matriz de confusión del ensemble
1667 cm = confusion_matrix(y_true, ens_bb, labels=labs)
1668 plt.figure(figsize=(7,6)); plt.imshow(cm, interpolation='nearest'); plt.title("CM -
1669     Ensemble BASE+BC (1/2/4)"); plt.colorbar()
1670 ticks=np.arange(len(labs)); plt.xticks(ticks, labs, rotation=45, ha='right'); plt.
1671     yticks(ticks, labs)
1672 for i in range(cm.shape[0]):
1673     for j in range(cm.shape[1]): plt.text(j,i,format(cm[i,j],'d'),ha="center",va=
1674         "center")
1675 plt.ylabel("Reales"); plt.xlabel("Predicciones")
1676 plt.tight_layout(); plt.savefig(vdir_ens / "cm_ensemble_BASEBC.png", dpi=160); plt.
1677     close()
1678
1679 # ----- 7) Guardados (CSV/JSON) -----
1680 pd.DataFrame({
1681     "y_true": y_true,
1682     "pred_rf_base": pred_base,
1683     "pred_ens_bb": ens_bb,
1684     "changed_bb": (pred_base!=ens_bb).astype(int)
1685 }).to_csv(vdir_ens / "predicciones_comparadas_test.csv", index=False)

```

```

1674
1675 pa_b.to_csv(vdir_ens / "per_class_rf_base.csv")
1676 pa_bb.to_csv(vdir_ens / "per_class_ensemble_BASEBC.csv")
1677
1678 with open(DIR_MODELOS / "ensemble_meta.json", "w", encoding="utf-8") as f:
1679     json.dump({
1680         "strategy": "rf_base_vs_rf124_basebc_flips_iterativos",
1681         "security_rules": {
1682             "global_not_worse": True,
1683             "class4_not_worse": True,
1684             "class1_drop_pp_max": 5
1685         },
1686         "margin_min": MARGIN_MIN,
1687         "uses": {
1688             "rf_base": "rf_base.pkl",
1689             "rf_124_basebc": "rf_124_basebc.pkl",
1690             "xgb_pack": "model_Noferta_XGB_v8.pkl"
1691         },
1692         "baja_comp": {"threshold": TH_BAJO, "duplicated_times": BAJA_COMP_DUP}
1693     }, f, indent=2, ensure_ascii=False)
1694
1695 with open(vdir_ens / "summary_ensemble.json", "w", encoding="utf-8") as f:
1696     json.dump({
1697         "acc_rf_base": float(acc_b), "macro_f1_rf_base": float(f1_b),
1698         "acc_ensemble_BASEBC": float(acc_bb), "macro_f1_ensemble_BASEBC": float(f1_bb)
1699     },
1700     "flips_BASEBC": int(len(flips_bb)),
1701     "ALLOWED_DROP_1": ALLOWED_DROP_1,
1702     "REQUIRE_IMPROVE_4": REQUIRE_IMPROVE_4,
1703     "REQUIRE_IMPROVE_GLOBAL": REQUIRE_IMPROVE_GLOBAL,
1704     "MARGIN_MIN": MARGIN_MIN,
1705     "baja_comp_dup": BAJA_COMP_DUP,
1706     "baja_comp_threshold": TH_BAJO
1707     }, f, indent=2, ensure_ascii=False)
1708
1709 with open(vdir_ens / "explicacion_tecnica.txt", "w", encoding="utf-8") as f:
1710     f.write(
1711         "RF_124_BASE+BC entrenado con MIN_PRE + 'baja_comp' (XGB) duplicada k=5. "
1712         "Ensemble 'seguro' vs RF base con flips iterativos por margen (≥0.03) y "
1713         "reglas de back-off: "
1714         "no empeorar accuracy global ni clase 4; clase 1 no cae >5 pp. "
1715         "Artefactos de producción en /modelos y validaciones en "
1716         "/validacion/06_ensemble_basebc."
1717     )
1718
1719 print("✅ Especialista y Ensemble BASE+BC listos. Artefactos en modelos/ y "
1720       "validacion/06_ensemble_basebc/")
1721
1722 """Celda E - META del RF_124_BASE+BC (MIN_PRE_USED + duplicadas BC exactas)"""
1723
1724 # =====
1725 # META - RF_124_BASE+BC (contrato de features)
1726 # =====
1727 import json, joblib, sklearn
1728 from pathlib import Path
1729
1730 BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
1731 DIR_MODELOS = BASE / "modelos"
1732
1733 rf124_path = DIR_MODELOS / "rf_124_basebc.pkl"
1734 assert rf124_path.exists(), "Falta modelos/rf_124_basebc.pkl"
1735 rf_124 = joblib.load(rf124_path)
1736
1737 def _cols_from_pipe(pipe):
1738     try:
1739         trfs = pipe.named_steps["prep"].transformers_
1740         cols = {}
1741         for name, _, c in pipe.named_steps["prep"].transformers_:
1742             cols[name] = list(c) if isinstance(c, list) else []
1743         return cols
1744     except Exception:
1745         return {"num": [], "cat": []}

```

```

1742 cols = _cols_from_pipe(rf_124)
1743
1744 # Detectar duplicadas de baja_comp exactamente como las vio el modelo
1745 dup_names = [c for c in (cols.get("cat", []) + cols.get("num", [])) if c.startswith(
1746 "baja_comp")]
1747 dup_k = max([int(c.replace("baja_comp_dup", "")) for c in dup_names if c != "baja_comp"
1748 ] or [0])
1749
1750 alias_mapping = globals().get("VAR_MAP", {}) # si existe
1751 rf124_meta = {
1752     "artifact": "rf_124_basebc.pkl",
1753     "expected_raw_features": cols,
1754     "alias_mapping_used_in_train": alias_mapping,
1755     "baja_comp_dup_k": dup_k,
1756     "baja_comp_dup_names": [c for c in dup_names if c != "baja_comp"],
1757     "requires_xgb_pack": "model_Noferta_XGB_v8.pkl",
1758     "sklearn_version": sklearn.__version__
1759 }
1760 (DIR_MODELOS / "rf_124_basebc_meta.json").write_text(json.dumps(rf124_meta, indent=2,
1761 ensure_ascii=False), encoding="utf-8")
1762 print("✅ Guardado modelos/rf_124_basebc_meta.json")
1763
1764 """Celda F – META del Ensemble (verificación)"""
1765
1766 # =====
1767 # META – Ensemble (verificación y dependencias)
1768 # =====
1769 import json
1770 from pathlib import Path
1771
1772 BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
1773 DIR_MODELOS = BASE / "modelos"
1774
1775 ens_path = DIR_MODELOS / "ensemble_meta.json"
1776 assert ens_path.exists(), "Falta modelos/ensemble_meta.json"
1777 ens = json.loads(ens_path.read_text(encoding="utf-8"))
1778
1779 ens.setdefault("depends_on", {})
1780 ens["depends_on"].update({
1781     "rf_base": "rf_base.pkl",
1782     "rf_124_basebc": "rf_124_basebc.pkl",
1783     "xgb_pack": "model_Noferta_XGB_v8.pkl",
1784     "preprocess": "preprocess_pipeline.pkl",
1785     "centroids": "centroides_pre.json"
1786 })
1787
1788 ens_path.write_text(json.dumps(ens, indent=2, ensure_ascii=False), encoding="utf-8")
1789 print("✅ Ensemble meta verificado/actualizado (modelos/ensemble_meta.json)")
1790
1791 """Celda G – Resumen rápido de contratos"""
1792
1793 # =====
1794 # META – Resumen de contratos guardados
1795 # =====
1796 from pathlib import Path
1797
1798 BASE = Path("/content/drive/MyDrive/_Pipeline_desarrollo_modelos_v2")
1799 DIR_MODELOS = BASE / "modelos"
1800 files = [
1801     "preprocess_meta.json",
1802     "knn_params.json",
1803     "knn_num_pipeline.joblib",
1804     "xgb_meta.json",
1805     "rf_base_meta.json",
1806     "rf_124_basebc_meta.json",
1807     "ensemble_meta.json"
1808 ]
1809 print("Contratos/artefactos meta en modelos/:")
1810 for f in files:
1811     p = DIR_MODELOS / f
1812     print("-", f, "OK" if p.exists() else "X")

```