```python
from __future__ import annotations

### SCRIPT 19 - PYTHON
# PIPELINE EJECUCION MODELOS EN GPT
# PIPELINE PREDICCION MODELOS DESDE ARCHIVOS JSON y NPZ
#
# ====================================================================

"""
Módulo de inferencia portable para el pipeline de riesgo en licitaciones.

- Lee artefactos JSON/NPZ desde un ZIP (modelos_portable.zip).
- Implementa versiones "puras" en NumPy/Pandas de:
  - Preprocesadores num+cat con imputación + escalado + one-hot.
  - RandomForestClassifier (RF base y RF_124_BASE+BC).
  - XGBoost multi-clase (BAJO/MEDIO/ALTO) para baja_comp.
  - kNN en el espacio PRE_NUM.
  - Ensemble RF base vs RF_124 con reglas simplificadas.

Notas importantes:
- No depende de scikit-learn ni xgboost.
- Asume una tolerancia práctica de ≈0,02 en predict_proba de RF
  respecto a sklearn; predict (las clases) debe coincidir si se respeta
  el contrato de entrada.
"""

import json
import math
import zipfile
from dataclasses import dataclass
from typing import Any, Dict, List, Optional, Tuple

import numpy as np
import pandas as pd


# ---------------------------------------------------------------------------
# Utilidades numéricas básicas
# ---------------------------------------------------------------------------

def _softmax(x: np.ndarray) -> np.ndarray:
    """
    Softmax fila a fila sobre un array 2D.

    Si se pasa un vector 1D, se convierte internamente en 2D (shape (1, n)).
    """
    x = np.asarray(x, dtype=float)
    if x.ndim == 1:
        x = x.reshape(1, -1)
    x_max = np.max(x, axis=1, keepdims=True)
    e = np.exp(x - x_max)
    s = e.sum(axis=1, keepdims=True)
    # Evitar divisiones por cero
    s[s == 0.0] = 1.0
    return e / s


def _ensure_2d_array(X: Any) -> np.ndarray:
    """
    Asegura que la entrada se convierte en un array 2D de tipo float.

    - Si es un DataFrame, usa .to_numpy().
    - Si es un vector 1D, lo convierte en shape (1, n_features).
    """
    if isinstance(X, pd.DataFrame):
        A = X.to_numpy(dtype=float)
    else:
        A = np.asarray(X, dtype=float)
    if A.ndim == 1:
        A = A.reshape(1, -1)
    return A

```

```python

def _is_int_string(s: str) -> bool:
    """Devuelve True si la cadena representa un entero (signo opcional)."""
    if s is None:
        return False
    s = str(s).strip()
    if not s:
        return False
    if s[0] in "+-":
        s = s[1:]
    return s.isdigit()

"""Celda 2- ZipStore"""

# ---------------------------------------------------------------------------
# ZipStore: lectura cómoda de JSON/NPZ desde un ZIP
# ---------------------------------------------------------------------------

@dataclass
class ZipStore:
    """
    Pequeño contenedor para acceder a artefactos dentro de un ZIP.

    Uso típico:
        store = ZipStore("/ruta/a/modelos_portable.zip")
        meta_rf = store.read_json("rf_base_portable.json")
        Z_train = store.read_npz("knn_train.npz")["Z"]

    Internamente construye un índice de:
        nombre_corto -> ruta_completa_en_el_zip

    por ejemplo:
        "rf_base_portable.json" -> "modelos_portable/rf_base_portable.json"
    """

    zip_path: str

    def __post_init__(self) -> None:
        # Abrimos el ZIP una única vez y construimos un índice por nombre corto
        self._zf = zipfile.ZipFile(self.zip_path, mode="r")
        self._index: Dict[str, str] = {}

        for name in self._zf.namelist():
            # Ignoramos directorios
            if name.endswith("/"):
                continue
            short = name.split("/")[-1]
            if short in self._index:
                # Preferimos fallar pronto si hay colisiones de nombres cortos
                raise ValueError(
                    f"Nombre corto duplicado en el ZIP: {short!r} "
                    f"(rutas: {self._index[short]!r} y {name!r})"
                )
            self._index[short] = name

    # ---------------- Métodos públicos de utilidad ----------------

    def list(self) -> List[str]:
        """Devuelve la lista de nombres cortos disponibles en el ZIP."""
        return sorted(self._index.keys())

    def has(self, short_name: str) -> bool:
        """Indica si existe un fichero con ese nombre corto en el ZIP."""
        return short_name in self._index

    def _full_name(self, short_name: str) -> str:
        """Resuelve el nombre corto a la ruta completa dentro del ZIP."""
        try:
            return self._index[short_name]
        except KeyError:
            raise FileNotFoundError(
                f"No se encuentra {short_name!r} en el ZIP {self.zip_path!r}."
```

```
            )

    def read_json(self, short_name: str) -> Dict[str, Any]:
        """
        Lee y decodifica un JSON dentro del ZIP.

        Parámetros
        ----------
        short_name : str
            Nombre corto del fichero, por ejemplo 'rf_base_portable.json'.

        Devuelve
        --------
        dict
            Contenido del JSON parseado.
        """
        full = self._full_name(short_name)
        raw = self._zf.read(full)
        return json.loads(raw.decode("utf-8"))

    def read_npz(self, short_name: str) -> np.lib.npyio.NpzFile:
        """
        Lee un fichero .npz dentro del ZIP y devuelve el objeto NpzFile.

        Parámetros
        ----------
        short_name : str
            Nombre corto del fichero, por ejemplo 'knn_train.npz'.

        Devuelve
        --------
        np.lib.npyio.NpzFile
            Objeto desde el que se pueden leer arrays, p.ej. ['Z'].
        """
        from io import BytesIO

        full = self._full_name(short_name)
        raw = self._zf.read(full)
        bio = BytesIO(raw)
        return np.load(bio)

    def close(self) -> None:
        """Cierra el ZIP subyacente (opcional; no es estrictamente necesario)."""
        try:
            self._zf.close()
        except Exception:
            pass

    # Soporte para usarlo como contexto: with ZipStore(...) as store:
    def __enter__(self) -> "ZipStore":
        return self

    def __exit__(self, exc_type, exc_val, exc_tb) -> None:
        self.close()

"""Celda 3 - Preprocesador portable (PortablePreprocessor)"""

# ---------------------------------------------------------------------------
# PortablePreprocessor
# ---------------------------------------------------------------------------

class PortablePreprocessor:
    """
    Preprocesador portable que reproduce:
      - SimpleImputer (num) + StandardScaler
      - SimpleImputer (cat) + OneHotEncoder(handle_unknown="ignore")

    El esquema debe tener la forma usada en:
      - preprocess_pre_portable.json
      - feature_space["preprocess"] de los RF portables.
    """
```

```python
    def __init__(self, schema: Dict[str, Any]) -> None:
        self.schema = schema

        num = schema.get("numeric", {}) or {}
        cat = schema.get("categorical", {}) or {}

        # ---------------- Configuración numérica ----------------
        self.num_cols: List[str] = list(num.get("columns", []))
        num_imp = num.get("imputer", {}) or {}
        num_scl = num.get("scaler", {}) or {}

        self.num_imp_strategy: str = num_imp.get("strategy", "median")
        self.num_imp_statistics: np.ndarray = np.asarray(
            num_imp.get("statistics", []), dtype=float
        )

        self.num_with_mean: bool = bool(num_scl.get("with_mean", True))
        self.num_with_std: bool = bool(num_scl.get("with_std", True))
        self.num_mean: np.ndarray = np.asarray(
            num_scl.get("mean", np.zeros(len(self.num_cols))), dtype=float
        )
        self.num_scale: np.ndarray = np.asarray(
            num_scl.get("scale", np.ones(len(self.num_cols))), dtype=float
        )
        # Evitar divisiones por cero más adelante
        self.num_scale[self.num_scale == 0.0] = 1.0

        # ---------------- Configuración categórica ----------------
        self.cat_cols: List[str] = list(cat.get("columns", []))
        cat_imp = cat.get("imputer", {}) or {}
        cat_oh = cat.get("onehot", {}) or {}

        self.cat_imp_strategy: str = cat_imp.get("strategy", "most_frequent")
        self.cat_fill_values: List[Any] = list(cat_imp.get("fill_values", []))
        self.cat_handle_unknown: str = cat_oh.get("handle_unknown", "ignore")

        # categories: dict col -> list[str]
        self.cat_categories: Dict[str, List[str]] = cat_oh.get("categories", {}) or {}

        # Nombres de salida opcionales (no imprescindibles para la inferencia)
        self.feature_names_out: Optional[List[str]] = schema.get("feature_names_out")

    # -------------------- numéricos --------------------

    def _transform_numeric(self, df: pd.DataFrame) -> np.ndarray:
        if not self.num_cols:
            return np.zeros((len(df), 0), dtype=float)

        X_num = pd.DataFrame(index=df.index)
        for c in self.num_cols:
            if c in df.columns:
                X_num[c] = pd.to_numeric(df[c], errors="coerce")
            else:
                # Columna ausente: todo NaN
                X_num[c] = np.nan

        A = X_num.to_numpy(dtype=float)  # shape (n_samples, n_features)

        # Imputación: implementamos 'median'/'most_frequent' vía statistics
        if self.num_imp_strategy in ("median", "most_frequent"):
            stats = self.num_imp_statistics
            if stats.shape[0] != A.shape[1]:
                raise ValueError(
                    "Dimensión de estadísticas numéricas inconsistente: "
                    f"{stats.shape[0]} != {A.shape[1]}"
                )
            # Imputamos NaN columna a columna
            for j in range(A.shape[1]):
                col = A[:, j]
                mask = np.isnan(col)
                if mask.any():
                    col[mask] = stats[j]
```

```
            # A ya modificado in-place

        # Escalado estándar
        if self.num_with_mean:
            A = A - self.num_mean
        if self.num_with_std:
            A = A / self.num_scale

        return A

    # -------------------- categóricas --------------------

    def _normalize_categorical_values(self, col: str, s: pd.Series) -> pd.Series:
        """
        Normaliza los valores de una columna categórica:

        - Si todas las categorías de esquema son enteros representados como str
          ("1","2",...), convierte los valores numéricos a int y luego a str
          (así 1, 1.0 y "1" se mapean a "1").
        - En otro caso, convierte a str tal cual.
        """
        cats = self.cat_categories.get(col, [])
        if cats and all(_is_int_string(c) for c in cats):
            # Columna con categorías "numéricas"
            def to_int_str(x: Any) -> Any:
                if pd.isna(x):
                    return np.nan  # se imputará antes
                sx = str(x).strip()
                try:
                    # Permite valores tipo "1.0"
                    val = int(float(sx))
                    return str(val)
                except Exception:
                    return sx  # fallback: lo que sea como str

            return s.map(to_int_str)
        else:
            # Columna categórica convencional
            return s.astype(str)

    def _transform_categorical(self, df: pd.DataFrame) -> np.ndarray:
        if not self.cat_cols:
            return np.zeros((len(df), 0), dtype=float)

        n = len(df)
        blocks: List[np.ndarray] = []

        for idx, col in enumerate(self.cat_cols):
            if col in df.columns:
                s = df[col]
            else:
                # Columna ausente: todo NaN
                s = pd.Series([np.nan] * n, index=df.index)

            # Imputación
            if self.cat_imp_strategy == "most_frequent":
                # fill_values está en el mismo orden que cat_cols
                if idx < len(self.cat_fill_values):
                    fill_val = self.cat_fill_values[idx]
                else:
                    # Fallback defensivo
                    fill_val = (
                        s.mode(dropna=True).iloc[0] if s.notna().any() else ""
                    )
                s = s.fillna(fill_val)
            else:
                # Otras estrategias no se contemplan en tus artefactos
                s = s.fillna("")

            # Normalización de valores (numéricos como enteros en str)
            s_norm = self._normalize_categorical_values(col, s)

```

```
            # One-hot
            cats = self.cat_categories.get(col, [])
            cats = [str(c) for c in cats]

            # Matriz (n_samples, len(cats))
            M = np.zeros((n, len(cats)), dtype=float)
            if len(cats) == 0:
                blocks.append(M)
                continue

            # Mapeo valor->índice de categoría
            cat_to_idx = {str(c): j for j, c in enumerate(cats)}

            # Para cada fila, activamos la posición correspondiente
            for i, val in enumerate(s_norm):
                if pd.isna(val):
                    # Ya imputado, pero por seguridad
                    continue
                key = str(val)
                j = cat_to_idx.get(key, None)
                if j is None:
                    # handle_unknown:
                    if self.cat_handle_unknown == "ignore":
                        continue
                    else:
                        # En tus modelos, esto no debería ocurrir; lanzamos error explícito
                        raise ValueError(
                            f"Valor desconocido '{key}' en columna '{col}' "
                            f"con handle_unknown='{self.cat_handle_unknown}'"
                        )
                M[i, j] = 1.0

            blocks.append(M)

        if blocks:
            return np.concatenate(blocks, axis=1)
        else:
            return np.zeros((len(df), 0), dtype=float)

    # -------------------- API pública --------------------

    def transform(self, df: pd.DataFrame) -> np.ndarray:
        """
        Aplica el preprocesado numérico + categórico y devuelve
        una matriz NumPy 2D (n_samples, n_features_transformados).
        """
        if not isinstance(df, pd.DataFrame):
            raise TypeError("PortablePreprocessor.transform espera un pandas.DataFrame.")

        A_num = self._transform_numeric(df)
        A_cat = self._transform_categorical(df)

        if A_num.shape[0] != A_cat.shape[0]:
            raise ValueError("Dimensiones numéricas y categóricas incompatibles.")

        if A_num.shape[1] == 0 and A_cat.shape[1] == 0:
            return np.zeros((len(df), 0), dtype=float)

        return np.hstack([A_num, A_cat])

"""Celda 4 con RandomForestPortable (RF Base y RF BC_124)"""

# ---------------------------------------------------------------------------
# RandomForestPortable
# ---------------------------------------------------------------------------

class RandomForestPortable:
    """
    Implementación portable de un RandomForestClassifier a partir de su JSON.

```

```
    - Usa PortablePreprocessor para replicar el preprocesado.
    - Cada árbol se evalúa explícitamente sobre la matriz transformada.
    """

    def __init__(self, artifact: Dict[str, Any]) -> None:
        """
        Parámetros
        ----------
        artifact : dict
            Diccionario cargado desde 'rf_base_portable.json' o
            'rf_124_portable.json'. Debe contener, al menos:

            - 'model_type': "RandomForestClassifier"
            - 'classes', 'n_classes'
            - 'feature_space': {
                'expected_raw_features': {'num': [...], 'cat': [...]},
                'alias_mapping_used_in_train': {...},
                'dtypes_expect': {...},
                'preprocess': {...}
              }
            - 'forest': {
                'n_estimators', 'max_depth', 'n_features_in', 'trees': [...]
              }
        """
        self.artifact = artifact
        if artifact.get("model_type") != "RandomForestClassifier":
            raise ValueError("Solo se soporta RandomForestClassifier en este
            portable.")

        # Clases y metadatos
        self.classes_: List[str] = [str(c) for c in artifact.get("classes", [])]
        self.n_classes_: int = int(artifact.get("n_classes", len(self.classes_)))

        # Espacio de características y preprocesador
        fs = artifact.get("feature_space", {}) or {}
        self.expected_raw_features: Dict[str, List[str]] = fs.get(
        "expected_raw_features", {}) or {}
        self.alias_mapping: Dict[str, str] = fs.get("alias_mapping_used_in_train", {})
         or {}
        self.dtypes_expect: Dict[str, str] = fs.get("dtypes_expect", {}) or {}
        prep_schema = fs.get("preprocess", {}) or {}
        self.preprocessor = PortablePreprocessor(prep_schema)

        # Bosque
        forest = artifact.get("forest", {}) or {}
        self.n_estimators: int = int(forest.get("n_estimators", 0))
        self.max_depth: Optional[int] = (
            int(forest["max_depth"]) if forest.get("max_depth") is not None else None
        )
        self.n_features_in_: int = int(forest.get("n_features_in", 0))

        self.trees: List[Dict[str, Any]] = list(forest.get("trees", []))
        if not self.trees:
            raise ValueError("El modelo RF portable no contiene árboles.")
        if self.n_classes_ <= 0:
            raise ValueError("El modelo RF portable no tiene clases definidas.")

    # -------------------- adaptador de entrada --------------------

    def _prepare_raw_input(self, df_raw: pd.DataFrame) -> pd.DataFrame:
        """
        Aplica alias de train y asegura que están todas las columnas
        de expected_raw_features.num y expected_raw_features.cat.

        - alias_mapping_used_in_train se interpreta como:
          {nombre_original_en_train: nombre_usado_en_modelo}
        """
        if not isinstance(df_raw, pd.DataFrame):
            raise TypeError("RandomForestPortable espera un pandas.DataFrame como
            entrada.")

        df = df_raw.copy()
```

```
499
500         # 1) Aliases de train: preferred -> used
501         #    alias_mapping_used_in_train viene típico como:
502         #    {"Criterio_precio_p": "C_precio_p", ...}
503         for preferred, used in self.alias_mapping.items():
504             if used in df.columns:
505                 # Ya existe la columna con el nombre usado en train
506                 continue
507             if preferred in df.columns:
508                 # Creamos la columna "used" copiando la "preferred"
509                 df[used] = df[preferred]
510
511         cols_num = list(self.expected_raw_features.get("num", []))
512         cols_cat = list(self.expected_raw_features.get("cat", []))
513         ordered = cols_num + cols_cat
514
515         # 2) Asegurar columnas y tipos mínimos
516         for c in ordered:
517             if c not in df.columns:
518                 # Columna esperada que no está: la creamos como NaN
519                 df[c] = np.nan
520
521         # Numéricas a float
522         for c in cols_num:
523             df[c] = pd.to_numeric(df[c], errors="coerce")
524
525         # Categóricas: no forzamos type, el PortablePreprocessor las tratará
526         return df[ordered]
527
528     # -------------------- evaluación de árboles --------------------
529
530     def _tree_predict_proba(self, tree: Dict[str, Any], X: np.ndarray) -> np.ndarray:
531         """
532         Evalúa un único árbol sobre todas las filas de X y devuelve
533         una matriz (n_samples, n_classes) con las probabilidades de ese árbol.
534         """
535         children_left = np.asarray(tree["children_left"], dtype=int)
536         children_right = np.asarray(tree["children_right"], dtype=int)
537         feature = np.asarray(tree["feature"], dtype=int)
538         threshold = np.asarray(tree["threshold"], dtype=float)
539         value = np.asarray(tree["value"], dtype=float)  # shape (n_nodes, n_classes)
540
541         n_nodes = int(tree.get("n_nodes", len(children_left)))
542         n_samples = X.shape[0]
543
544         out = np.zeros((n_samples, self.n_classes_), dtype=float)
545
546         for i in range(n_samples):
547             node = 0
548             # Descenso por el árbol
549             while True:
550                 # Nodo fuera de rango: por robustez, volvemos a la raíz y salimos
551                 if node < 0 or node >= n_nodes:
552                     node = 0
553                     break
554
555                 # Consideramos hoja si no tiene hijos o si la feature es < 0
556                 if (
557                     children_left[node] == -1
558                     and children_right[node] == -1
559                 ) or feature[node] < 0:
560                     break
561
562                 f = feature[node]
563                 thr = threshold[node]
564                 x_ij = X[i, f]
565
566                 if x_ij <= thr:
567                     node = children_left[node]
568                 else:
569                     node = children_right[node]
570
```

```
            if node < 0 or node >= n_nodes:
                # Nodo inválido: por robustez, usamos la raíz
                node = 0

            proba_node = value[node]
            # Re-normalizamos por seguridad
            s = proba_node.sum()
            if s <= 0.0:
                proba_node = np.ones(self.n_classes_, dtype=float) / float(self.
                n_classes_)
            else:
                proba_node = proba_node / s

            out[i, :] = proba_node

        return out

    # -------------------- API pública --------------------

    def predict_proba(self, df_raw: pd.DataFrame) -> np.ndarray:
        """
        Devuelve la matriz de probabilidades (n_samples, n_classes),
        promediando las predicciones de todos los árboles.
        """
        df_prep = self._prepare_raw_input(df_raw)
        X = self.preprocessor.transform(df_prep)
        X = _ensure_2d_array(X)

        if X.shape[1] != self.n_features_in_ and self.n_features_in_ > 0:
            # Aviso suave; no lanzamos error duro para permitir cierta tolerancia
            pass

        n_samples = X.shape[0]
        proba_acc = np.zeros((n_samples, self.n_classes_), dtype=float)

        for tree in self.trees:
            proba_acc += self._tree_predict_proba(tree, X)

        # Promedio de todos los árboles
        proba_acc /= float(len(self.trees))

        # Normalización final por seguridad
        sums = proba_acc.sum(axis=1, keepdims=True)
        sums[sums == 0.0] = 1.0
        proba_acc /= sums

        return proba_acc

    def predict(self, df_raw: pd.DataFrame) -> np.ndarray:
        """Devuelve las clases predichas (np.ndarray de str)."""
        proba = self.predict_proba(df_raw)
        idx = np.argmax(proba, axis=1)
        classes = np.array(self.classes_, dtype=object)
        return classes[idx]

"""Celda 5 - XGBBajaPortable (XGBoost multi-clase, umbral BAJO fijo 0,38)"""

# ------------------------------------------------------------------------
# XGBBajaPortable
# ------------------------------------------------------------------------

class XGBBajaPortable:
    """
    Implementación portable del modelo XGBoost multi-clase (BAJO/MEDIO/ALTO)
    para predecir baja_comp.

    - Objetivo: multi:softprob.
    - A partir de xgb_baja_portable.json + xgb_baja_model.json.
    """

    def __init__(self, meta: Dict[str, Any], booster: Dict[str, Any]) -> None:
        self.meta = meta
```

```
642         self.variables_predictoras: List[str] = list(meta.get("variables_predictoras",
              []))
643         self.encoders_schema: Dict[str, Dict[str, Any]] = meta.get("encoders_schema",
            {}) or {}
644         self.labels: List[str] = list(meta.get("labels", [])) or ["BAJO", "MEDIO",
            "ALTO"]
645         self.policies: Dict[str, Any] = meta.get("policies", {}) or {}
646         self.fallback_class_por_col: Dict[str, str] = self.policies.get(
647             "fallback_class_por_col", {}
648         )
649
650         # Umbral fijo para la clase BAJO, independientemente del meta
651         # (decisión explícita para este script portable).
652         self.threshold_bajo: float = 0.38
653
654         # Booster JSON (estructura interna de XGBoost)
655         learner = booster.get("learner", {}) or {}
656         self.feature_names: List[str] = list(learner.get("feature_names", []))
657         lm_param = learner.get("learner_model_param", {}) or {}
658         self.base_score: float = float(lm_param.get("base_score", 0.5))
659         self.num_class: int = int(lm_param.get("num_class", len(self.labels) or 3))
660
661         gb = learner.get("gradient_booster", {}) or {}
662         model = gb.get("model", {}) or {}
663         self.trees: List[Dict[str, Any]] = list(model.get("trees", []))
664         self.tree_info: List[int] = list(model.get("tree_info", []))
665
666         if len(self.tree_info) != len(self.trees):
667             raise ValueError("Inconsistencia entre número de árboles y tree_info en
                XGB.")
668
669         # Índice de la clase BAJO
670         try:
671             self.idx_bajo: int = self.labels.index("BAJO")
672         except ValueError:
673             self.idx_bajo = 0
674
675         # Preconstruimos esquemas de encoders: col -> {clase: índice}
676         self._encoders_index: Dict[str, Dict[str, int]] = {}
677         for col, schema in self.encoders_schema.items():
678             classes = [str(c) for c in schema.get("classes", [])]
679             self._encoders_index[col] = {cls: i for i, cls in enumerate(classes)}
680
681     # -------------------- preprocesado de entrada --------------------
682
683     def _prepare_X(self, df_raw: pd.DataFrame) -> pd.DataFrame:
684         """
685         Construye el DataFrame X con las variables_predictoras en el orden
686         exacto esperado por el booster.
687         """
688         if not isinstance(df_raw, pd.DataFrame):
689             raise TypeError("XGBBajaPortable espera un pandas.DataFrame como entrada."
                )
690
691         X = pd.DataFrame(index=df_raw.index)
692
693         # 1) Columnas básicas (num y cat ya codificados como en df_norm)
694         for col in self.variables_predictoras:
695             if col in df_raw.columns:
696                 X[col] = df_raw[col]
697             else:
698                 # Política de columna numérica faltante: fill_0 (coherente con meta)
699                 X[col] = 0
700
701         # 2) Encoders categóricos tipo LabelEncoder
702         fallback_map = self.fallback_class_por_col or {}
703         for col, idx_map in self._encoders_index.items():
704             if col in X.columns:
705                 s = X[col].astype(str)
706                 classes = set(idx_map.keys())
707                 fb = str(fallback_map.get(col, next(iter(classes)) if classes else ""
                    ))
```

```
                # Sustituimos valores fuera del repertorio por el fallback
                s = s.where(s.isin(classes), other=fb)

                # Transformamos a índices
                def to_idx(val: Any) -> int:
                    try:
                        return idx_map[str(val)]
                    except KeyError:
                        return idx_map.get(fb, 0)

                X[col] = s.map(to_idx).astype(int)
            else:
                # Columna prevista pero ausente -> 0
                X[col] = 0

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

        return X[self.variables_predictoras]

    # -------------------- evaluación de árboles --------------------

    def _tree_predict_margin(self, tree: Dict[str, Any], X: np.ndarray) -> np.ndarray:
        """
        Evalúa un árbol XGBoost sobre X y devuelve los márgenes (una dimensión).
        """
        left_children = np.asarray(tree["left_children"], dtype=int)
        right_children = np.asarray(tree["right_children"], dtype=int)
        split_indices = np.asarray(tree["split_indices"], dtype=int)
        split_conditions = np.asarray(tree["split_conditions"], dtype=float)
        default_left = np.asarray(tree["default_left"], dtype=int)
        base_weights = np.asarray(tree["base_weights"], dtype=float)

        n_nodes = int(tree.get("tree_param", {}).get("num_nodes", len(left_children)))
        n_samples = X.shape[0]

        out = np.zeros(n_samples, dtype=float)

        for i in range(n_samples):
            node = 0
            while node >= 0 and node < n_nodes:
                left = left_children[node]
                right = right_children[node]

                # Hoja: no hay hijos
                if left == -1 and right == -1:
                    break

                feat_idx = split_indices[node]
                thr = split_conditions[node]
                v = X[i, feat_idx]

                if math.isnan(v):
                    # Usamos default_left
                    go_left = bool(default_left[node])
                else:
                    # Regla típica de XGBoost: v < thr -> izquierda
                    go_left = v < thr

                node = left if go_left else right
                if node == -1:
                    break

            if node == -1:
                # Por robustez, volvemos a la raíz
```

```
                node = 0

            out[i] = base_weights[node]

        return out

    # -------------------- API pública --------------------

    def predict_proba(self, df_raw: pd.DataFrame) -> np.ndarray:
        """
        Devuelve la matriz de probabilidades (n_samples, num_class)
        del XGBoost multi-clase (BAJO/MEDIO/ALTO).
        """
        X_df = self._prepare_X(df_raw)
        X = X_df.to_numpy(dtype=float)
        X = _ensure_2d_array(X)

        n_samples = X.shape[0]
        margins = np.full((n_samples, self.num_class), self.base_score, dtype=float)

        for t_idx, tree in enumerate(self.trees):
            class_id = int(self.tree_info[t_idx])
            if class_id < 0 or class_id >= self.num_class:
                continue
            contrib = self._tree_predict_margin(tree, X)
            margins[:, class_id] += contrib

        # Usamos el softmax fila a fila definido en las utilidades
        proba = _softmax(margins)
        return proba

    def predict_baja(self, df_raw: pd.DataFrame) -> Tuple[pd.Series, pd.Series]:
        """
        Devuelve:
          - baja_comp: Series de int (0/1)
          - p_bajo: Series de float (probabilidad de BAJO)
        """
        proba = self.predict_proba(df_raw)
        if proba.shape[1] <= self.idx_bajo:
            # Modelo mal formado; devolvemos todo ceros
            p_bajo = np.zeros(proba.shape[0], dtype=float)
        else:
            p_bajo = proba[:, self.idx_bajo]

        baja_flag = (p_bajo >= self.threshold_bajo).astype(np.int8)
        idx = df_raw.index
        return (
            pd.Series(baja_flag, index=idx, name="baja_comp"),
            pd.Series(p_bajo, index=idx, name="p_bajo"),
        )

"""Celda 6 - KNNPortable (espacio PRE_NUM, distancias y similitud)"""

# ---------------------------------------------------------------------------
# KNNPortable
# ---------------------------------------------------------------------------

class KNNPortable:
    """
    Implementación portable del espacio kNN PRE_NUM.

    Se basa en:
      - knn_space_portable.json
      - knn_train.npz (Z_train)
      - knn_all.npz   (Z_all)

    Funcionalidad:
      - Proyectar nuevas filas al espacio PRE_NUM con el pipeline numérico
        (imputación + escalado).
      - Calcular distancias euclídeas a Z_train o Z_all.
      - Devolver índices y distancias de los k vecinos, junto con una
        medida de similitud (%) basada en p95_nn1_train.
```

```
    """

    def __init__(
        self,
        meta: Dict[str, Any],
        Z_train: np.ndarray,
        Z_all: np.ndarray,
    ) -> None:
        self.meta = meta
        self.pre_num_features: List[str] = list(meta.get("pre_num_features", []))
        self.metric: str = str(meta.get("metric", "euclidean"))
        self.k_neighbors: int = int(meta.get("k_neighbors", 25))
        self.p95_nn1_train: float = float(meta.get("p95_nn1_train", 0.0))

        num_pipeline = meta.get("num_pipeline", {}) or {}
        num_imp = num_pipeline.get("imputer", {}) or {}
        num_scl = num_pipeline.get("scaler", {}) or {}

        self.imp_strategy: str = num_imp.get("strategy", "median")
        self.imp_statistics: np.ndarray = np.asarray(
            num_imp.get("statistics", []), dtype=float
        )

        self.with_mean: bool = bool(num_scl.get("with_mean", True))
        self.with_std: bool = bool(num_scl.get("with_std", True))
        self.mean_: np.ndarray = np.asarray(
            num_scl.get("mean", np.zeros(len(self.pre_num_features))), dtype=float
        )
        self.scale_: np.ndarray = np.asarray(
            num_scl.get("scale", np.ones(len(self.pre_num_features))), dtype=float
        )
        self.scale_[self.scale_ == 0.0] = 1.0  # evitar divisiones por cero

        # Catálogo (metadatos informativos; no podemos reconstruir el DF desde aquí)
        self.catalog_meta: Dict[str, Any] = meta.get("catalog", {}) or {}

        # Matrices de referencia
        self.Z_train: np.ndarray = np.asarray(Z_train, dtype=float)
        self.Z_all: np.ndarray = np.asarray(Z_all, dtype=float)

        if self.Z_train.ndim != 2 or self.Z_all.ndim != 2:
            raise ValueError("Z_train y Z_all deben ser matrices 2D.")

    # -------------------- proyección al espacio PRE_NUM --------------------

    def _build_X_num(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Construye el DataFrame numérico X_num con las columnas pre_num_features
        en el orden exacto, aplicando algunas reglas de fallback coherentes
        con el pipeline original (C_precio_p <- Criterio_precio_p).
        """
        if not isinstance(df, pd.DataFrame):
            raise TypeError("KNNPortable espera un pandas.DataFrame como entrada.")

        X_num = pd.DataFrame(index=df.index)

        for c in self.pre_num_features:
            if c in df.columns:
                X_num[c] = pd.to_numeric(df[c], errors="coerce")
            elif c == "C_precio_p" and "Criterio_precio_p" in df.columns:
                # Fallback: usar la columna original de criterio si existe
                X_num[c] = pd.to_numeric(df["Criterio_precio_p"], errors="coerce")
            else:
                X_num[c] = np.nan

        return X_num[self.pre_num_features]

    def _transform(self, df: pd.DataFrame) -> np.ndarray:
        """
        Aplica el imputer + scaler numérico del pipeline kNN a df, y devuelve
        la matriz NumPy 2D en el espacio PRE_NUM.
        """
```

```
            X_num = self._build_X_num(df)
            A = X_num.to_numpy(dtype=float)

            # Imputación por estadísticos
            if self.imp_strategy in ("median", "most_frequent"):
                stats = self.imp_statistics
                if stats.shape[0] != A.shape[1]:
                    raise ValueError(
                        "Dimensión de estadísticas de imputación kNN inconsistente: "
                        f"{stats.shape[0]} != {A.shape[1]}"
                    )
                for j in range(A.shape[1]):
                    col = A[:, j]
                    mask = np.isnan(col)
                    if mask.any():
                        col[mask] = stats[j]

            # Escalado
            if self.with_mean:
                A = A - self.mean_
            if self.with_std:
                A = A / self.scale_

            return A

        # -------------------- distancias y vecinos --------------------

        def _compute_distances(self, Zq: np.ndarray, space: str) -> np.ndarray:
            """
            Calcula la matriz de distancias entre Zq (consultas) y Z_base
            (train o all) usando la métrica configurada.

            Actualmente solo se implementa 'euclidean'; otras métricas se
            degradan a euclidiana de forma explícita.
            """
            if space == "train":
                Z_base = self.Z_train
            elif space == "all":
                Z_base = self.Z_all
            else:
                raise ValueError("space debe ser 'train' o 'all'.")

            if self.metric not in ("euclidean", "l2"):
                # Degradamos a euclidiana, lo documentamos en el docstring
                pass

            # Broadcasting: (n_q, 1, d) - (1, n_ref, d) -> (n_q, n_ref, d)
            diff = Zq[:, None, :] - Z_base[None, :, :]
            dist = np.sqrt(np.sum(diff ** 2, axis=2))  # shape (n_q, n_ref)
            return dist

        def neighbors(
            self,
            df: pd.DataFrame,
            topk: int = 10,
            space: str = "all",
        ) -> Dict[str, Any]:
            """
            Calcula los k vecinos más cercanos para cada fila de df.

            Parámetros
            ----------
            df : DataFrame
                DataFrame con las variables PRE necesarias (al menos las que
                figuran en pre_num_features).
            topk : int, por defecto 10
                Número de vecinos a devolver (se trunca si hay menos filas
                en Z_base).
            space : {"train","all"}, por defecto "all"
                Conjunto de referencia (Z_train o Z_all).

            Devuelve
```

```
        --------
        dict con:
            - "indices" : np.ndarray (n_samples, k) de índices de vecinos.
            - "distances": np.ndarray (n_samples, k) de distancias euclídeas.
            - "similarity_percent": np.ndarray (n_samples, k) con sim% = max(0, 1 -
            d/p95)*100.
            - "p95_nn1_train": float (el mismo valor meta).
        """
        Zq = self._transform(df)
        Zq = _ensure_2d_array(Zq)

        dist = self._compute_distances(Zq, space=space)
        n_q, n_ref = dist.shape

        k = max(1, min(int(topk), n_ref))
        # argsort a lo largo del eje de referencia
        # índices de vecinos ordenados por distancia ascendente
        idx_sorted = np.argsort(dist, axis=1)[:, :k]

        # Distancias ordenadas
        rows = np.arange(n_q)[:, None]
        dist_sorted = dist[rows, idx_sorted]

        # Similitud porcentual basada en p95_nn1_train
        if self.p95_nn1_train > 0:
            sim_percent = np.maximum(
                0.0,
                1.0 - dist_sorted / float(self.p95_nn1_train),
            ) * 100.0
        else:
            sim_percent = np.zeros_like(dist_sorted)

        return {
            "indices": idx_sorted.astype(int),
            "distances": dist_sorted.astype(float),
            "similarity_percent": sim_percent.astype(float),
            "p95_nn1_train": float(self.p95_nn1_train),
        }

"""Celda 7 - EnsemblePredictor (nueva lógica RF base vs RF_124)"""

# ---------------------------------------------------------------------------
# EnsemblePredictor
# ---------------------------------------------------------------------------

class EnsemblePredictor:
    """
    Ensemble simplificado entre RF base y RF_124_BASE+BC.

    Lógica:
      - Tomamos RF base como predicción por defecto.
      - Solo activamos el ensemble cuando la clase top-1 del RF base está
        en classes_124 = {"1","2","4"}.
      - Además exigimos que la clase top-1 del RF_124 también esté en ese
        mismo conjunto.
      - En ese caso, comparamos la probabilidad top-1 (p1) de cada modelo
        y escogemos el modelo cuya p1 sea mayor.
      - En caso de empate, se mantiene la predicción del RF base.
      - margin_min y otras reglas "seguras" NO se aplican aquí.

    La salida incorpora también la información de baja_comp del modelo XGB.
    """

    def __init__(self, classes_124: Optional[List[str]] = None) -> None:
        if classes_124 is None:
            classes_124 = ["1", "2", "4"]
        # Guardamos el conjunto de clases donde se considera el especialista
        self.classes_124 = set(str(c) for c in classes_124)

    # ------------------- utilidades internas --------------------

    @staticmethod
```

```python
    def _top1_top2(row: np.ndarray, labels: List[str]) -> Tuple[str, float, float]:
        """
        Dada una fila de probabilidades y la lista de labels,
        devuelve (label_top1, p1, p2), donde p2 es la segunda probabilidad
        más alta (útil si quisieras volver a un criterio por margen).
        """
        row = np.asarray(row, dtype=float)
        if row.ndim != 1:
            raise ValueError("_top1_top2 espera un vector 1D de probabilidades.")

        # Ordenamos índices por probabilidad descendente
        idx_sorted = np.argsort(row)[::-1]
        idx1 = idx_sorted[0]
        p1 = float(row[idx1])

        if len(idx_sorted) > 1:
            idx2 = idx_sorted[1]
            p2 = float(row[idx2])
        else:
            p2 = 0.0

        label_top1 = str(labels[idx1])
        return label_top1, p1, p2

    # -------------------- API principal --------------------

    def apply(
        self,
        pred_base: Any,
        proba_base: Any,
        labels_base: List[str],
        pred_124: Any,
        proba_124: Any,
        labels_124: List[str],
        baja_comp: pd.Series,
        p_bajo: pd.Series,
    ) -> pd.DataFrame:
        """
        Aplica la lógica de ensemble sobre las predicciones de RF base,
        RF_124 y XGB baja.

        Parámetros
        ----------
        pred_base : array-like
            Clases predichas por RF base (no se usan directamente aquí,
            se recalculan desde las probabilidades).
        proba_base : array-like (n_samples, n_classes_base)
            Probabilidades de RF base.
        labels_base : list[str]
            Etiquetas de las clases de RF base en el mismo orden que las columnas
            de proba_base.
        pred_124 : array-like
            Clases predichas por RF_124 (no se usan directamente aquí).
        proba_124 : array-like (n_samples, n_classes_124)
            Probabilidades de RF_124.
        labels_124 : list[str]
            Etiquetas de las clases de RF_124 en el mismo orden que las columnas
            de proba_124.
        baja_comp : pd.Series
            Indicador 0/1 de baja_comp procedente de XGB.
        p_bajo : pd.Series
            Probabilidad de clase BAJO procedente de XGB.

        Devuelve
        --------
        DataFrame con columnas:
          - 'pred'             : clase final (str)
          - 'confianza_pred'   : probabilidad top-1 del modelo elegido
          - 'fuente_confianza' : 'rf_base' o 'rf_124_basebc'
          - 'flip_ensemble'    : bool, True si se cambió desde rf_base a rf_124
          - 'baja_comp'        : 0/1
          - 'p_bajo'           : probabilidad de BAJO
```

```
        """
        # Convertimos a arrays NumPy
        proba_base = np.asarray(proba_base, dtype=float)
        proba_124 = np.asarray(proba_124, dtype=float)

        n = proba_base.shape[0]
        if proba_124.shape[0] != n:
            raise ValueError("proba_base y proba_124 tienen número de filas distinto."
            )

        # Aseguramos que baja_comp y p_bajo tienen longitud compatible
        if len(baja_comp) != n or len(p_bajo) != n:
            raise ValueError(
                "baja_comp y p_bajo deben tener la misma longitud que las matrices de
                probas."
            )

        # Index para el DataFrame de salida
        idx_out = baja_comp.index

        # Inicializamos salidas
        pred_final = np.empty(n, dtype=object)
        fuente = np.full(n, "rf_base", dtype=object)
        conf_final = np.zeros(n, dtype=float)
        flip = np.zeros(n, dtype=bool)

        # Recorremos fila a fila
        for i in range(n):
            row_base = proba_base[i]
            row_124 = proba_124[i]

            base_label_top1, p1_base, _ = self._top1_top2(row_base, labels_base)
            spec_label_top1, p1_spec, _ = self._top1_top2(row_124, labels_124)

            # Por defecto, siempre usamos RF base
            pred_final[i] = base_label_top1
            conf_final[i] = p1_base
            fuente[i] = "rf_base"

            # Solo activamos ensemble si la clase base está en {1,2,4}
            if base_label_top1 not in self.classes_124:
                continue

            # Y si RF_124 también predice una clase válida del especialista
            if spec_label_top1 not in self.classes_124:
                continue

            # Regla de decisión: elegir el modelo con mayor probabilidad top-1
            if p1_spec > p1_base:
                pred_final[i] = spec_label_top1
                conf_final[i] = p1_spec
                fuente[i] = "rf_124_basebc"
                flip[i] = True
            # Si p1_spec <= p1_base, nos quedamos con la predicción base

        df_out = pd.DataFrame(
            {
                "pred": pred_final.astype(str),
                "confianza_pred": conf_final.astype(float),
                "fuente_confianza": fuente.astype(str),
                "flip_ensemble": flip.astype(bool),
                "baja_comp": baja_comp.astype(int).values,
                "p_bajo": p_bajo.astype(float).values,
            },
            index=idx_out,
        )

        return df_out

"""Celda 8 - PortablePredictor + load_predictor_from_zip (API de alto nivel)"""

# ------------------------------------------------------------------------
```

```python
# PortablePredictor: fachada principal
# ---------------------------------------------------------------------------

class PortablePredictor:
    """
    Fachada principal para usar todos los modelos portables desde un ZIP.

    Flujo típico de uso:
      predictor = PortablePredictor.from_zip("modelos_portable.zip")
      resultados = predictor.predict(df_norm)
      vecinos = predictor.knn_neighbors(df_norm.iloc[[0]], topk=10, space="all")

    Importante:
      - Este predictor asume que df_norm (df_raw en las firmas) ya está en el
        formato "normalizado" que consumen los modelos:
          * columnas y tipos compatibles con expected_raw_features de los RF
          * variables_predictoras del XGB de baja_comp
          * pre_num_features del espacio kNN
        Es decir, df_norm debería ser la salida del script de transformación
        (export_python_transformacion.py) aplicada sobre los datos originales.
    """

    def __init__(
        self,
        zip_path: str,
        store: ZipStore,
        rf_base: RandomForestPortable,
        rf_124: RandomForestPortable,
        xgb_baja: XGBBajaPortable,
        knn: KNNPortable,
        ensemble: EnsemblePredictor,
        baja_comp_dup_times: int,
        ensemble_meta: Dict[str, Any],
        preprocess_pre: Optional[PortablePreprocessor] = None,
        centroids_pre: Optional[Dict[str, Any]] = None,
    ) -> None:
        self.zip_path = zip_path
        self.store = store
        self.rf_base = rf_base
        self.rf_124 = rf_124
        self.xgb_baja = xgb_baja
        self.knn = knn
        self.ensemble = ensemble
        self.baja_comp_dup_times = int(baja_comp_dup_times)
        self.ensemble_meta = ensemble_meta
        self.preprocess_pre = preprocess_pre
        self.centroids_pre = centroids_pre

    # -------------------- construcción desde ZIP --------------------

    @classmethod
    def from_zip(cls, zip_path: str) -> "PortablePredictor":
        """
        Carga todos los artefactos portables desde un ZIP y construye
        un PortablePredictor listo para usar.

        Parámetros
        ----------
        zip_path : str
            Ruta al fichero modelos_portable.zip.

        Devuelve
        --------
        PortablePredictor
        """
        store = ZipStore(zip_path)

        # 1) Meta del ensemble (punto de entrada común)
        if not store.has("ensemble_portable.json"):
            raise FileNotFoundError(
                "No se ha encontrado 'ensemble_portable.json' en el ZIP."
            )
```

```
        ensemble_meta = store.read_json("ensemble_portable.json")

        deps = ensemble_meta.get("depends_on_portable", {}) or {}

        # 2) Artefactos RF base y RF_124
        rf_base_name = deps.get("rf_base", "rf_base_portable.json")
        rf_124_name = deps.get("rf_124_basebc", "rf_124_portable.json")

        if not store.has(rf_base_name):
            raise FileNotFoundError(f"Falta artefacto RF base: {rf_base_name!r}")
        if not store.has(rf_124_name):
            raise FileNotFoundError(f"Falta artefacto RF_124: {rf_124_name!r}")

        rf_base_art = store.read_json(rf_base_name)
        rf_124_art = store.read_json(rf_124_name)

        rf_base = RandomForestPortable(rf_base_art)
        rf_124 = RandomForestPortable(rf_124_art)

        # 3) Artefacto XGB baja_comp (meta + booster)
        xgb_meta_name = deps.get("xgb_baja", "xgb_baja_portable.json")
        if not store.has(xgb_meta_name):
            raise FileNotFoundError(f"Falta artefacto XGB meta: {xgb_meta_name!r}")
        xgb_meta = store.read_json(xgb_meta_name)

        # Booster guardado con Booster.save_model en formato JSON
        if not store.has("xgb_baja_model.json"):
            raise FileNotFoundError("Falta 'xgb_baja_model.json' en el ZIP.")
        xgb_booster = store.read_json("xgb_baja_model.json")

        xgb_baja = XGBBajaPortable(xgb_meta, xgb_booster)

        # 4) Espacio kNN PRE_NUM
        knn_meta_name = deps.get("knn_space", "knn_space_portable.json")
        if not store.has(knn_meta_name):
            raise FileNotFoundError(f"Falta artefacto kNN meta: {knn_meta_name!r}")
        knn_meta = store.read_json(knn_meta_name)

        if not store.has("knn_train.npz"):
            raise FileNotFoundError("Falta 'knn_train.npz' en el ZIP.")
        if not store.has("knn_all.npz"):
            raise FileNotFoundError("Falta 'knn_all.npz' en el ZIP.")

        Z_train_npz = store.read_npz("knn_train.npz")
        Z_all_npz = store.read_npz("knn_all.npz")

        if "Z" not in Z_train_npz.files:
            raise KeyError("En 'knn_train.npz' se esperaba un array 'Z'.")
        if "Z" not in Z_all_npz.files:
            raise KeyError("En 'knn_all.npz' se esperaba un array 'Z'.")

        knn = KNNPortable(
            meta=knn_meta,
            Z_train=Z_train_npz["Z"],
            Z_all=Z_all_npz["Z"],
        )

        # 5) Preprocesador PRE global (opcional, por si quieres usarlo fuera)
        preprocess_pre = None
        centroids_pre = None
        pre_pre_name = deps.get("preprocess_pre", "preprocess_pre_portable.json")
        if store.has(pre_pre_name):
            pre_pre_art = store.read_json(pre_pre_name)
            # El esquema de preprocess_pre es compatible con PortablePreprocessor
            prep_schema = {
                "numeric": pre_pre_art.get("numeric", {}),
                "categorical": pre_pre_art.get("categorical", {}),
                "feature_names_out": pre_pre_art.get("feature_names_out"),
            }
            preprocess_pre = PortablePreprocessor(prep_schema)
            centroids_pre = pre_pre_art.get("centroids_ref") or pre_pre_art.get(
            "centroids")
```

```
1351
1352            # 6) Ensemble
1353            ensemble = EnsemblePredictor(classes_124=["1", "2", "4"])
1354
1355            # 7) Configuración de baja_comp duplicada para RF_124
1356            baja_cfg = ensemble_meta.get("baja_comp", {}) or {}
1357            baja_comp_dup_times = int(baja_cfg.get("duplicated_times", 5))
1358
1359            return cls(
1360                zip_path=zip_path,
1361                store=store,
1362                rf_base=rf_base,
1363                rf_124=rf_124,
1364                xgb_baja=xgb_baja,
1365                knn=knn,
1366                ensemble=ensemble,
1367                baja_comp_dup_times=baja_comp_dup_times,
1368                ensemble_meta=ensemble_meta,
1369                preprocess_pre=preprocess_pre,
1370                centroids_pre=centroids_pre,
1371            )
1372
1373        # -------------------- utilidades internas --------------------
1374
1375        @staticmethod
1376        def _ensure_dataframe(df: Any) -> pd.DataFrame:
1377            """
1378            Asegura que la entrada se convierte en un DataFrame.
1379
1380            - Si ya es DataFrame, se devuelve una copia.
1381            - Si es dict, se interpreta como un único registro (index = [0]).
1382            - Si es Series, se interpreta como una fila (index = [0]).
1383            """
1384            if isinstance(df, pd.DataFrame):
1385                return df.copy()
1386            elif isinstance(df, pd.Series):
1387                return df.to_frame().T.copy()
1388            elif isinstance(df, dict):
1389                return pd.DataFrame([df]).copy()
1390            else:
1391                raise TypeError(
1392                    "PortablePredictor espera un DataFrame, Series o dict como entrada."
1393                )
1394
1395        def _attach_baja_comp_dups_for_rf124(
1396            self,
1397            df: pd.DataFrame,
1398            baja_comp: pd.Series,
1399        ) -> pd.DataFrame:
1400            """
1401            Inserta 'baja_comp' y sus duplicadas en df para el RF_124, siguiendo
1402            la información de expected_raw_features del modelo portable.
1403
1404            Estrategia:
1405              - Columna base: 'baja_comp' (si no existe, se crea).
1406              - Duplicadas: todas las columnas categóricas de RF_124 cuyo nombre
1407                empiece por 'baja_comp_dup'.
1408            """
1409            df2 = df.copy()
1410
1411            # Base
1412            df2["baja_comp"] = baja_comp.astype(int)
1413
1414            # Duplicadas según expected_raw_features del RF_124
1415            exp_124 = self.rf_124.artifact.get("feature_space", {}).get(
                "expected_raw_features", {}) or {}
1416            cat_124 = list(exp_124.get("cat", []))
1417            dup_names = [c for c in cat_124 if c.startswith("baja_comp_dup")]
1418
1419            for dn in dup_names:
1420                df2[dn] = df2["baja_comp"]
1421
```

```
1422            return df2
1423
1424        # -------------------- API pública: predicciones --------------------
1425
1426        def predict(self, df_raw: Any) -> Dict[str, Any]:
1427            """
1428            Ejecuta toda la cadena de predicción:
1429
1430              - XGB baja_comp -> baja_comp (0/1) + p_bajo
1431              - RF base       -> clases + probabilidades
1432              - RF_124        -> clases + probabilidades (usando baja_comp y duplicadas)
1433              - Ensemble      -> combinación RF base vs RF_124 + baja_comp
1434
1435            Parámetros
1436            ----------
1437            df_raw : DataFrame, Series o dict
1438                df_norm con todas las columnas necesarias para los modelos.
1439
1440            Devuelve
1441            --------
1442            dict con:
1443              - "rf_base" : {"pred": Series, "proba": DataFrame}
1444              - "rf_124"  : {"pred": Series, "proba": DataFrame}
1445              - "xgb_baja": {"baja_comp": Series, "p_bajo": Series, "threshold_bajo":
                  float}
1446              - "ensemble": DataFrame con la predicción final y metadatos.
1447            """
1448            df = self._ensure_dataframe(df_raw)
1449
1450            # 1) XGB baja_comp
1451            baja_comp, p_bajo = self.xgb_baja.predict_baja(df)
1452
1453            # 2) RF base
1454            proba_base = self.rf_base.predict_proba(df)
1455            pred_base = self.rf_base.predict(df)
1456
1457            # 3) RF_124 (requiere baja_comp + duplicadas)
1458            df_rf124 = self._attach_baja_comp_dups_for_rf124(df, baja_comp)
1459            proba_124 = self.rf_124.predict_proba(df_rf124)
1460            pred_124 = self.rf_124.predict(df_rf124)
1461
1462            # 4) Ensemble
1463            ensemble_df = self.ensemble.apply(
1464                pred_base=pred_base,
1465                proba_base=proba_base,
1466                labels_base=self.rf_base.classes_,
1467                pred_124=pred_124,
1468                proba_124=proba_124,
1469                labels_124=self.rf_124.classes_,
1470                baja_comp=baja_comp,
1471                p_bajo=p_bajo,
1472            )
1473
1474            # Construimos salidas en estructuras cómodas
1475            idx = df.index
1476            rf_base_out = {
1477                "pred": pd.Series(pred_base, index=idx, name="rf_base_pred"),
1478                "proba": pd.DataFrame(proba_base, index=idx, columns=self.rf_base.classes_
                    ),
1479            }
1480            rf_124_out = {
1481                "pred": pd.Series(pred_124, index=idx, name="rf_124_pred"),
1482                "proba": pd.DataFrame(proba_124, index=idx, columns=self.rf_124.classes_),
1483            }
1484            xgb_out = {
1485                "baja_comp": baja_comp,
1486                "p_bajo": p_bajo,
1487                "threshold_bajo": float(self.xgb_baja.threshold_bajo),
1488            }
1489
1490            return {
1491                "rf_base": rf_base_out,
```

```
            "rf_124": rf_124_out,
            "xgb_baja": xgb_out,
            "ensemble": ensemble_df,
        }

    def knn_neighbors(
        self,
        df: Any,
        topk: int = 10,
        space: str = "all",
    ) -> Dict[str, Any]:
        """
        Calcula vecinos más cercanos en el espacio PRE_NUM a partir de df.

        Parámetros
        ----------
        df : DataFrame, Series o dict
            df_norm con, al menos, las columnas pre_num_features del kNN.
        topk : int
            Número máximo de vecinos a devolver.
        space : {"train","all"}
            Conjunto de referencia (Z_train o Z_all).

        Devuelve
        --------
        dict como el de KNNPortable.neighbors().
        """
        df2 = self._ensure_dataframe(df)
        return self.knn.neighbors(df2, topk=topk, space=space)


# ---------------------------------------------------------------------------
# Función de conveniencia
# ---------------------------------------------------------------------------

def load_predictor_from_zip(zip_path: str) -> PortablePredictor:
    """
    Punto de entrada de alto nivel para construir un PortablePredictor
    a partir de un ZIP de artefactos portables.
    """
    return PortablePredictor.from_zip(zip_path)


# ---------------------------------------------------------------------------
# Ejemplo de uso (comentado)
# ---------------------------------------------------------------------------

# Ejemplo de cómo usarlo en tu entorno (descomentar y adaptar rutas):
#
# zip_path =
"/content/drive/MyDrive/_Pipeline_produccion_prediccion/modelos_portable.zip"
# predictor = load_predictor_from_zip(zip_path)
#
# import pandas as pd
# df_test = pd.read_excel(
#     "/content/drive/MyDrive/_Pipeline_produccion_prediccion/df_test_train_v8.xlsx"
# )
# # Supongamos que df_test ya se ha transformado con export_python_transformacion.py
# # para obtener df_norm; aquí usarías df_norm en lugar de df_test si lo tienes:
# df_norm = df_test  # sustitúyelo por tu df_norm real
#
# resultados = predictor.predict(df_norm.head(5))
# print(resultados["ensemble"].head())
# vecinos = predictor.knn_neighbors(df_norm.iloc[[0]], topk=10, space="all")
# print(vecinos["indices"][0], vecinos["distances"][0])

"""Celda 9 - Helper build_prediction_payload + ejemplo JSON"""

# ---------------------------------------------------------------------------
# Celda 9: construcción de un payload JSON explicable para un expediente
# ---------------------------------------------------------------------------
```

```python
import json

def build_prediction_payload(
    predictor: PortablePredictor,
    df_raw: pd.DataFrame,
    row_index: int = 0,
    knn_topk: int = 10,
    knn_space: str = "all",
) -> Dict[str, Any]:
    """
    Construye un diccionario (JSON-serializable) con toda la información
    relevante para un único expediente.

    Pensado para que un GPT (con Python) pueda:
      - ejecutar esta función,
      - obtener el dict,
      - y a partir de ahí generar una explicación en lenguaje natural.

    Parámetros
    ----------
    predictor : PortablePredictor
        Instancia ya cargada con load_predictor_from_zip().
    df_raw : DataFrame
        DataFrame con al menos una fila, en el formato esperado por predictor.
        (idealmente, df_norm tras export_python_transformacion).
    row_index : int
        Índice posicional (iloc) de la fila a explicar.
    knn_topk : int
        Número de vecinos más cercanos a devolver.
    knn_space : {"train","all"}
        Conjunto de referencia para el kNN.

    Devuelve
    --------
    dict
        Objeto listo para pasarse a json.dumps().
    """
    if not isinstance(df_raw, pd.DataFrame):
        raise TypeError("build_prediction_payload espera un pandas.DataFrame en
        df_raw.")

    if df_raw.shape[0] == 0:
        raise ValueError("df_raw no contiene filas.")

    if row_index < 0 or row_index >= df_raw.shape[0]:
        raise IndexError(f"row_index={row_index} fuera de rango para df_raw con {
        df_raw.shape[0]} filas.")

    # Extraemos una sola fila como DataFrame (preserva nombres de columnas)
    df_one = df_raw.iloc[[row_index]].copy()
    idx_label = df_one.index[0]

    # 1) Predicciones de todos los modelos
    res = predictor.predict(df_one)

    ensemble_df = res["ensemble"]
    rf_base_out = res["rf_base"]
    rf_124_out = res["rf_124"]
    xgb_out = res["xgb_baja"]

    # Fila única del ensemble
    ens_row = ensemble_df.loc[idx_label]

    # RF base y RF_124: proba de la fila
    rf_base_proba_row = rf_base_out["proba"].loc[idx_label]
    rf_124_proba_row = rf_124_out["proba"].loc[idx_label]

    # 2) Vecinos kNN para esa misma fila
    knn_res = predictor.knn_neighbors(df_one, topk=knn_topk, space=knn_space)
    knn_indices = knn_res["indices"][0].tolist()
    knn_distances = [float(d) for d in knn_res["distances"][0]]
    knn_similarity = [float(s) for s in knn_res["similarity_percent"][0]]
```

```
    p95_nn1_train = float(knn_res["p95_nn1_train"])

    # 3) Entrada "cruda": valores de la fila en df_raw (ya normalizado)
    input_row = df_one.iloc[0].to_dict()
    # Convertimos posibles tipos numpy a tipos nativos
    input_row_clean = {}
    for k, v in input_row.items():
        if isinstance(v, (np.generic, np.bool_)):
            input_row_clean[k] = v.item()
        else:
            input_row_clean[k] = v

    # 4) Construimos el payload
    payload = {
        "input": {
            "index_label": str(idx_label),
            "row_index": int(row_index),
            "features": input_row_clean,
        },
        "models": {
            "xgb_baja": {
                "p_bajo": float(xgb_out["p_bajo"].loc[idx_label]),
                "baja_comp": int(xgb_out["baja_comp"].loc[idx_label]),
                "threshold_bajo": float(xgb_out["threshold_bajo"]),
                "labels": list(predictor.xgb_baja.labels),
            },
            "rf_base": {
                "pred": str(rf_base_out["pred"].loc[idx_label]),
                "proba": {
                    str(cls): float(rf_base_proba_row[cls])
                    for cls in predictor.rf_base.classes_
                },
                "classes": [str(c) for c in predictor.rf_base.classes_],
            },
            "rf_124": {
                "pred": str(rf_124_out["pred"].loc[idx_label]),
                "proba": {
                    str(cls): float(rf_124_proba_row[cls])
                    for cls in predictor.rf_124.classes_
                },
                "classes": [str(c) for c in predictor.rf_124.classes_],
            },
        },
        "ensemble": {
            "pred_final": str(ens_row["pred"]),
            "confianza_pred": float(ens_row["confianza_pred"]),
            "fuente_confianza": str(ens_row["fuente_confianza"]),
            "flip_ensemble": bool(ens_row["flip_ensemble"]),
            "baja_comp": int(ens_row["baja_comp"]),
            "p_bajo": float(ens_row["p_bajo"]),
        },
        "knn": {
            "space": str(knn_space),
            "topk": int(knn_topk),
            "indices": [int(i) for i in knn_indices],
            "distances": knn_distances,
            "similarity_percent": knn_similarity,
            "p95_nn1_train": p95_nn1_train,
        },
        "meta": {
            "zip_path": str(predictor.zip_path),
            "notes": (
                "Probabilidades RF portables pueden diferir hasta ~0.02 respecto "
                "a sklearn; threshold_bajo del XGB se fija explícitamente en 0.38."
            ),
        },
    }

    return payload


# Ejemplo de uso rápido (puedes adaptarlo a tu df_norm real):
```

```
#
# zip_path =
"/content/drive/MyDrive/_Pipeline_produccion_prediccion/modelos_portable.zip"
# predictor = load_predictor_from_zip(zip_path)
#
# import pandas as pd
# df_test =
pd.read_excel("/content/drive/MyDrive/_Pipeline_produccion_prediccion/df_test_train_v8
.xlsx")
# df_norm = df_test  # sustituye aquí por df_norm si ya lo tienes transformado
#
# payload = build_prediction_payload(predictor, df_norm, row_index=0, knn_topk=10,
knn_space="all")
# print(json.dumps(payload, ensure_ascii=False, indent=2))

"""Celda 11 – Documento RAG completo por expediente"""

# ---------------------------------------------------------------------------
# Celda 11: documento RAG completo por expediente
#           (JSON-safe + vecinos kNN + summary + risk_label + meta.version)
# ---------------------------------------------------------------------------

import json
from pathlib import Path
import math
import datetime
from typing import Any, Dict

def _to_jsonable_scalar(v: Any) -> Any:
    """
    Convierte un valor escalar a un tipo JSON-serializable.

    - np.generic / np.bool_      -> .item()
    - pandas.Timestamp           -> ISO8601 (str)
    - datetime.date/datetime     -> ISO8601 (str)
    - pandas.Timedelta           -> str (segundos)
    - float NaN                  -> None (se convierte en null en JSON)
    - Resto                      -> se devuelve tal cual
    """
    # Números y booleanos numpy (np.generic, np.bool_)
    if isinstance(v, (np.generic, np.bool_)):
        v = v.item()

    # Fechas/horas de pandas
    if isinstance(v, pd.Timestamp):
        return v.isoformat()

    # Fechas/horas estándar
    if isinstance(v, (datetime.datetime, datetime.date)):
        return v.isoformat()

    # Duraciones
    if isinstance(v, pd.Timedelta):
        return str(v.total_seconds())

    # NaN como null (JSON estándar)
    if isinstance(v, float) and math.isnan(v):
        return None

    return v


def build_rag_case_document(
    predictor: PortablePredictor,
    df_raw: pd.DataFrame,
    row_index: int = 0,
    knn_topk: int = 10,
    knn_space: str = "all",
) -> Dict[str, Any]:
    """
    Construye un documento RAG v4 para un expediente concreto,
    listo para ser indexado o usado directamente por un GPT.
```

```
        Estructura principal:
          - doc_type        : "licitacion_riesgo_prediccion"
          - rag_metodologia : info sobre la metodología RAG v4 (Source, etc.)
          - source          : payload factual (predicciones, kNN, entrada)
          - retriever       : campos clave para filtrado/búsqueda en el índice
          - evaluation      : placeholders para métricas/feedback futuro

        Además:
          - 'source.knn.neighbors' contiene ya los datos completos de los casos
            vecinos (filas de df_raw), incluyendo un bloque 'summary' con las
            6 variables principales:
              * Ahorro_final
              * Presupuesto_licitacion_lote_c
              * N_ofertantes
              * Plazo_m
              * Baja_p
              * C_precio_p
          - 'source.models.xgb_baja' incorpora:
              * 'proba': probabilidades por clase (BAJO/MEDIO/ALTO)
              * 'risk_label': etiqueta final de riesgo de baja competencia
          - 'source.meta' incorpora:
              * 'model_set_version'
              * 'generated_at' (UTC, ISO8601)
        """
        # 1) Payload factual base (una fila + kNN)
        payload = build_prediction_payload(
            predictor=predictor,
            df_raw=df_raw,
            row_index=row_index,
            knn_topk=knn_topk,
            knn_space=knn_space,
        )

        # --- Meta: versión de modelos y timestamp de generación ---
        meta_payload = payload.get("meta", {}) or {}
        meta_payload["model_set_version"] = "v1.0_portable_rag"
        meta_payload["generated_at"] = datetime.datetime.utcnow().isoformat() + "Z"
        payload["meta"] = meta_payload

        # --- Riesgo XGB: proba completa + risk_label ---
        xgb_block = payload["models"]["xgb_baja"]
        # Recalculamos las probabilidades completas para las 3 clases del XGB
        df_one = df_raw.iloc[[row_index]].copy()
        proba_all = predictor.xgb_baja.predict_proba(df_one)  # shape (1, num_class)
        proba_vec = proba_all[0]
        labels_list = list(predictor.xgb_baja.labels)

        # Lista python de probabilidades para trabajar cómodamente
        proba_list = [float(p) for p in proba_vec]
        # Guardamos el vector completo en el bloque XGB
        xgb_block["proba"] = {
            str(lbl): float(proba_list[i]) for i, lbl in enumerate(labels_list)
        }

        # Definimos risk_label:
        #   - Si baja_comp = 1 y existe clase "BAJO" -> "BAJO"
        #   - Si baja_comp = 0 -> clase con mayor probabilidad entre el resto
        #     (MEDIO/ALTO, etc.); si no hay "BAJO", usamos argmax global.
        baja_flag = int(xgb_block["baja_comp"])
        if "BAJO" in labels_list:
            idx_bajo = labels_list.index("BAJO")
        else:
            idx_bajo = None

        if baja_flag == 1 and idx_bajo is not None:
            risk_label = "BAJO"
        else:
            if idx_bajo is not None and len(labels_list) > 1:
                # Candidatas: todas menos BAJO
                cand = [
                    (lbl, proba_list[i])
```

```
                    for i, lbl in enumerate(labels_list)
                    if i != idx_bajo
                ]
                if cand:
                    risk_label = max(cand, key=lambda t: t[1])[0]
                else:
                    # Fallback: argmax global
                    idx_max = max(range(len(proba_list)), key=lambda i: proba_list[i])
                    risk_label = labels_list[idx_max]
            else:
                # Fallback: argmax global
                idx_max = max(range(len(proba_list)), key=lambda i: proba_list[i])
                risk_label = labels_list[idx_max]

        xgb_block["risk_label"] = str(risk_label)
        payload["models"]["xgb_baja"] = xgb_block

        # --- Limpieza JSON-safe de las features de entrada ---
        features_raw = payload["input"]["features"]
        features_clean = {k: _to_jsonable_scalar(v) for k, v in features_raw.items()}
        payload["input"]["features"] = features_clean

        # Identificadores básicos del caso
        index_label = payload["input"]["index_label"]
        row_idx = payload["input"]["row_index"]
        ensemble = payload["ensemble"]
        rf_base = payload["models"]["rf_base"]
        rf_124 = payload["models"]["rf_124"]

        # 2) Enriquecer la sección kNN con datos completos de vecinos + summary
        knn_section = payload.get("knn", {}) or {}
        indices = knn_section.get("indices", []) or []
        neighbors_list = []
        n_rows = df_raw.shape[0]

        for rank, idx in enumerate(indices):
            # Índice en Z_all (y, asumimos, en df_raw)
            try:
                idx_int = int(idx)
            except Exception:
                continue

            if idx_int < 0 or idx_int >= n_rows:
                # Por robustez, si el índice está fuera de rango
                neighbors_list.append(
                    {
                        "rank": int(rank + 1),
                        "knn_index": idx_int,
                        "case_id": None,
                        "summary": None,
                        "features": None,
                    }
                )
                continue

            # Fila completa del vecino en df_raw
            row = df_raw.iloc[idx_int]
            row_dict = row.to_dict()
            row_clean = {k: _to_jsonable_scalar(v) for k, v in row_dict.items()}

            # Bloque summary con las 6 variables principales
            summary = {
                "Ahorro_final": row_clean.get("Ahorro_final"),
                "Presupuesto_licitacion_lote_c": row_clean.get(
                "Presupuesto_licitacion_lote_c"),
                "N_ofertantes": row_clean.get("N_ofertantes"),
                "Plazo_m": row_clean.get("Plazo_m"),
                "Baja_p": row_clean.get("Baja_p"),
                "C_precio_p": row_clean.get("C_precio_p"),
            }

            neighbor_entry = {
```

```
                "rank": int(rank + 1),
                "knn_index": idx_int,
                # Intentamos usar el Identificador PLACSP como ID de caso; si no está, usamos el índice
                "case_id": _to_jsonable_scalar(row_dict.get("Identificador", idx_int)),
                "summary": summary,
                "features": row_clean,
            }
            neighbors_list.append(neighbor_entry)

        # Insertamos los vecinos en la sección kNN
        knn_section["neighbors"] = neighbors_list
        payload["knn"] = knn_section

        # 3) Bloque retriever: campos útiles para indexar y filtrar en el RAG
        retriever = {
            # Identificador primario (índice del DataFrame)
            "case_id": index_label,
            "row_index": row_idx,
            # Identificadores "de negocio" (PLACSP) si existen
            "business_ids": {
                "Identificador": features_clean.get("Identificador"),
                "Identificador_lote": features_clean.get("Identificador_lote"),
                "Identificador_lici": features_clean.get("Identificador_lici_c"),
            },
            # Etiquetas de salida clave
            "labels": {
                "pred_ensemble": ensemble["pred_final"],
                "baja_comp_flag": ensemble["baja_comp"],
                "p_bajo": ensemble["p_bajo"],
                "rf_base_pred": rf_base["pred"],
                "rf_124_pred": rf_124["pred"],
                "xgb_risk_label": xgb_block["risk_label"],
            },
            # Algunas features clave para filtrado rápido
            "key_features": {
                k: v
                for k, v in features_clean.items()
                if k in {
                    "Presupuesto_licitacion_lote_c",
                    "N_ofertantes",
                    "Plazo_m",
                    "Baja_p",
                    "C_precio_p",
                    "C_juicio_valor_p_c",
                    "Mes_lici",
                    "Tipo_de_Administracion_c",
                    "Tipo_de_procedimiento_c",
                    "Provincia2",
                    "CPV_c",
                    "CPV_lote_c",
                }
            },
            # Lista completa de campos disponibles para filtrado/consulta avanzada
            "available_feature_keys": list(features_clean.keys()),
        }

        # 4) Metadatos de metodología RAG v4 (informativos)
        rag_metodologia = {
            "version": "v4",
            "modules": [
                {
                    "name": "Source",
                    "role": (
                        "Preparar y procesar datos estructurados de predicción, "
                        "incluyendo entradas, salidas de modelos y casos similares."
                    ),
                },
                {
                    "name": "UI",
                    "role": (
                        "Interfaz de usuario que mostrará al licitador la explicación "
```

```
                    "basada en este documento (fuera de este script)."
                ),
            },
            {
                "name": "Orchestration",
                "role": (
                    "Coordinar la carga de este JSON, la invocación al GPT y el "
                    "almacenamiento de feedback (gestionado fuera de este script)."
                ),
            },
            {
                "name": "Retriever",
                "role": (
                    "Recuperar documentos relevantes usando las claves de 'retriever', "
                    "por ejemplo por pred_ensemble, baja_comp_flag o CPV."
                ),
            },
            {
                "name": "Generator",
                "role": (
                    "Generar la explicación en lenguaje natural a partir de este "
                    "documento y otras fuentes (tesis, normativa, etc.)."
                ),
            },
            {
                "name": "Reranker",
                "role": (
                    "Reordenar documentos recuperados si hay múltiples casos "
                    "candidatos para una consulta dada."
                ),
            },
            {
                "name": "Evaluation",
                "role": (
                    "Medir y registrar la calidad de las respuestas generadas "
                    "usando este documento como base factual."
                ),
            },
        ],
    }

    # 5) Placeholder para evaluación (se rellenará en otro módulo del sistema)
    evaluation = {
        "human_feedback": None,
        "notes": None,
        "metrics": {
            "retrieval": None,
            "generation": None,
        },
    }

    # 6) Documento RAG completo
    rag_doc = {
        "doc_type": "licitacion_riesgo_prediccion",
        "rag_metodologia": rag_metodologia,
        "source": payload,          # payload ya con features, vecinos y XGB enriquecido
        "retriever": retriever,
        "evaluation": evaluation,
    }

    return rag_doc


def generate_and_save_rag_case_document(
    predictor: PortablePredictor,
    df_raw: pd.DataFrame,
    row_index: int,
    knn_topk: int = 10,
    knn_space: str = "all",
    output_path: str = "rag_case_doc.json",
```

```python
) -> Dict[str, Any]:
    """
    Genera el documento RAG completo para un expediente y lo guarda en JSON.

    Este será el archivo 'legible y utilizable' por el GPT:
      - Se puede cargar con json.load(...)
      - El GPT lo usará como documento fuente en el RAG, sin necesidad de
        acceder a df_norm ni a otros artefactos de kNN.
    """
    rag_doc = build_rag_case_document(
        predictor=predictor,
        df_raw=df_raw,
        row_index=row_index,
        knn_topk=knn_topk,
        knn_space=knn_space,
    )

    output_path = Path(output_path)
    with output_path.open("w", encoding="utf-8") as f:
        json.dump(rag_doc, f, ensure_ascii=False, indent=2)

    print(f"Documento RAG guardado en {output_path.resolve()}")
    return rag_doc
```