```
PIPELINE AUDIT - SCRIPTS NO EJECUTADOS (STATIC REACHABILITY)
Repo: C:\Data-LakeHouse
Src : C:\Data-LakeHouse\src
Settings: C:\Data-LakeHouse\config\settings.yaml
Total scripts NO usados: 65

Listado:
\src\audit\audit_ajuste_poscosecha_ml2.py
\src\audit\audit_desp_poscosecha_ml2.py
\src\audit\audit_dh_poscosecha_ml2.py
\src\audit\audit_harvest_horizon_ml2.py
\src\audit\audit_harvest_start_ml2.py
\src\audit\audit_hidr_poscosecha_ml2.py
\src\audit\audit_ml1_curva_share_vs_real.py
\src\audit\audit_ml1_curva_shares.py
\src\audit\audit_peso_tallo_ml2.py
\src\audit\audit_poscosecha_ml2_baseline.py
\src\audit\audit_share_grado_ml2.py
\src\audit\audit_tallos_curve_ml2.py
\src\bronze\build_conteo_tallos_alturas_gyp2419_raw.py
\src\bronze\build_fenograma_raw.py
\src\eval\build_kpis_ajuste_poscosecha_ml2.py
\src\eval\build_kpis_desp_poscosecha_ml2.py
\src\eval\build_kpis_dh_poscosecha_ml2.py
\src\eval\build_kpis_harvest_horizon_ml2.py
\src\eval\build_kpis_harvest_start_ml2.py
\src\eval\build_kpis_hidr_poscosecha_ml2.py
\src\eval\build_kpis_peso_tallo_ml2.py
\src\eval\build_kpis_poscosecha_ml2_baseline.py
\src\eval\build_kpis_share_grado_ml2.py
\src\eval\build_kpis_tallos_curve_ml2.py
\src\gold\build_ds_ajuste_poscosecha_ml2_v1.py
\src\gold\build_ds_desp_poscosecha_ml2_v1.py
\src\gold\build_ds_dh_poscosecha_ml2_v1.py
\src\gold\build_ds_harvest_horizon_ml2_v2.py
\src\gold\build_ds_harvest_start_ml2_v2.py
\src\gold\build_ds_hidr_poscosecha_ml2_v1.py
\src\gold\build_ds_tallos_curve_ml2_v2.py
\src\gold\build_pred_kg_cajas_ml2.py
\src\gold\build_pred_poscosecha_ml2_final_views.py
\src\gold\build_pred_poscosecha_ml2_seed_mix_grado_dia.py
\src\gold\build_pred_tallos_ml2_full.py
\src\gold\postprocess_curva_share_smooth_ml1.py
\src\models\ml1\apply_curva_cdf_dia.py
\src\models\ml1\apply_curva_share_dia.py
\src\models\ml1\apply_curva_tallos_dia.py
\src\models\ml1\train_ajuste_poscosecha_ml1.py
\src\models\ml1\train_curva_beta_multiplier_dia.py
\src\models\ml1\train_curva_beta_params.py
\src\models\ml1\train_curva_cdf_dia.py
\src\models\ml1\train_curva_share_dia.py
\src\models\ml1\train_curva_tallos_dia.py
\src\models\ml1\train_desp_poscosecha_ml1.py
\src\models\ml1\train_dh_poscosecha_ml1.py
\src\models\ml1\train_dist_grado.py
\src\models\ml1\train_harvest_window_ml1.py
\src\models\ml1\train_hidr_poscosecha_ml1.py
\src\models\ml1\train_peso_tallo_grado.py
\src\models\ml2\apply_ajuste_poscosecha_ml2.py
\src\models\ml2\apply_desp_poscosecha_ml2.py
\src\models\ml2\train_ajuste_poscosecha_ml2.py
\src\models\ml2\train_desp_poscosecha_ml2.py
\src\models\ml2\train_dh_poscosecha_ml2.py
\src\models\ml2\train_harvest_horizon_ml2.py
\src\models\ml2\train_harvest_start_ml2.py
\src\models\ml2\train_hidr_poscosecha_ml2.py
\src\models\ml2\train_peso_tallo_ml2.py
\src\models\ml2\train_share_grado_ml2.py
\src\models\ml2\train_tallos_curve_ml2.py
\src\preds\build_pred_oferta_dia_from_universe_ml1.py
\src\silver\build_ciclo_maestro_from_fenograma.py
\src\silver\build_windows_from_milestones_final.py
```

```
------------------------------------------------------
[1/65] FILE: \src\audit\audit_ajuste_poscosecha_ml2.py
------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"

IN_G = EVAL / "ml2_ajuste_poscosecha_eval_global.parquet"
IN_D = EVAL / "ml2_ajuste_poscosecha_eval_by_destino.parquet"
IN_DIST = EVAL / "ml2_ajuste_poscosecha_eval_ratio_dist.parquet"

# Re-lectura de final para ejemplos
GOLD = DATA / "gold"
SILVER = DATA / "silver"
IN_FINAL = GOLD / "pred_poscosecha_ml2_ajuste_grado_dia_bloque_destino_final.parquet"
IN_REAL_MA = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"


def _ts() -> str:
    return datetime.utcnow().strftime("%Y%m%d_%H%M%S")


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def main() -> None:
    ts = _ts()

    out_g = EVAL / f"audit_ajuste_poscosecha_ml2_kpi_global_{ts}.parquet"
    out_d = EVAL / f"audit_ajuste_poscosecha_ml2_kpi_by_destino_{ts}.parquet"
    out_dist = EVAL / f"audit_ajuste_poscosecha_ml2_ratio_dist_{ts}.parquet"
    out_ex = EVAL / f"audit_ajuste_poscosecha_ml2_examples_10x2_{ts}.parquet"

    g = read_parquet(IN_G).copy()
    d = read_parquet(IN_D).copy()
    dist = read_parquet(IN_DIST).copy()

    # Examples: top improve / worst by |log_ratio_ml1|-|log_ratio_ml2|
    df = read_parquet(IN_FINAL).copy()
    df.columns = [str(c).strip() for c in df.columns]
    df["destino"] = _canon_str(df["destino"])

    # real
    real = read_parquet(IN_REAL_MA).copy()
    real.columns = [str(c).strip() for c in real.columns]
    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    if "factor_ajuste" in real.columns:
        rc = "factor_ajuste"
    elif "ajuste" in real.columns:
        rc = "ajuste"
    else:
        rc = None
    if rc is None:
        raise ValueError("Real MA no trae factor_ajuste/ajuste.")
    real[rc] = pd.to_numeric(real[rc], errors="coerce")
    real2 = real.groupby(["fecha_post", "destino"], dropna=False, as_index=False).agg(factor_ajuste_real=(rc,
"median"))

    # columnas
    fecha_post_col = None
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:
        if c in df.columns:
            fecha_post_col = c
            break
    if fecha_post_col is None:
        raise KeyError("No encuentro fecha_post_pred_* en final.")
    df[fecha_post_col] = _to_date(df[fecha_post_col])

    aj_ml1_col = None
    for c in ["factor_ajuste_ml1", "ajuste_ml1", "factor_ajuste_seed", "factor_ajuste"]:
        if c in df.columns:
            aj_ml1_col = c
            break
    if aj_ml1_col is None:
        raise KeyError("No encuentro ajuste ML1 en final.")
    df[aj_ml1_col] = pd.to_numeric(df[aj_ml1_col], errors="coerce")

    df["factor_ajuste_final"] = pd.to_numeric(df["factor_ajuste_final"], errors="coerce")

    df = df.merge(
        real2.rename(columns={"fecha_post": "fecha_post_key"}),
        left_on=[fecha_post_col, "destino"],
        right_on=["fecha_post_key", "destino"],
        how="left",
    )

    m = df["factor_ajuste_real"].notna() & df[aj_ml1_col].notna() & df["factor_ajuste_final"].notna()
    x = df.loc[m].copy()
    if x.empty:
        # igual guardamos los KPI y dist
        EVAL.mkdir(parents=True, exist_ok=True)
        write_parquet(g, out_g)
        write_parquet(d, out_d)
        write_parquet(dist, out_dist)
        write_parquet(pd.DataFrame([]), out_ex)
        print("\n=== ML2 AJUSTE POSCOSECHA AUDIT ===")
        print(f"KPI global parquet : {out_g}")
        print(f"KPI por destino    : {out_d}")
        print(f"Ratio dist parquet : {out_dist}")
        print(f"Examples 10x2      : {out_ex}")
        print("\n[WARN] No hay filas con real para examples.")
        return

    # pesos (si existen)
    w = None
    for c in ["tallos_w", "tallos", "tallos_total_ml2", "tallos_total"]:
        if c in x.columns:
            w = pd.to_numeric(x[c], errors="coerce").fillna(0.0)
            break
    if w is None:
        w = pd.Series(1.0, index=x.index, dtype="float64")

    ratio_ml1 = (x[aj_ml1_col] / x["factor_ajuste_real"].replace(0, np.nan)).astype(float)
    ratio_ml2 = (x["factor_ajuste_final"] / x["factor_ajuste_real"].replace(0, np.nan)).astype(float)

    lr1 = np.log(ratio_ml1.replace(0, np.nan))
    lr2 = np.log(ratio_ml2.replace(0, np.nan))
    score = lr1.abs() - lr2.abs()   # >0 mejora
    wscore = score * w

    x["ratio_ml1"] = ratio_ml1
    x["ratio_ml2"] = ratio_ml2
    x["log_ratio_ml1"] = lr1
    x["log_ratio_ml2"] = lr2
    x["score"] = score
    x["wscore"] = wscore

    cols = [
        "fecha",
        fecha_post_col,
        "destino",
        "grado" if "grado" in x.columns else None,
        aj_ml1_col,
        "factor_ajuste_final",
        "factor_ajuste_real",
        "ratio_ml1",
        "ratio_ml2",
        "log_ratio_ml1",
        "log_ratio_ml2",
        "score",
        "wscore",
    ]
    cols = [c for c in cols if c is not None and c in x.columns]
    top = x.sort_values("wscore", ascending=False).head(10).copy()
    top["sample_group"] = "TOP_IMPROVE_10"
    worst = x.sort_values("wscore", ascending=True).head(10).copy()
    worst["sample_group"] = "TOP_WORSE_10"
    ex = pd.concat([top[cols + ["sample_group"]], worst[cols + ["sample_group"]]], ignore_index=True)

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(g, out_g)
    write_parquet(d, out_d)
    write_parquet(dist, out_dist)
    write_parquet(ex, out_ex)

    print("\n=== ML2 AJUSTE POSCOSECHA AUDIT ===")
    print(f"KPI global parquet : {out_g}")
    print(f"KPI por destino    : {out_d}")
    print(f"Ratio dist parquet : {out_dist}")
    print(f"Examples 10x2      : {out_ex}")
    print("\n--- KPI GLOBAL ---")
    print(g.to_string(index=False))
    print("\n--- RATIO DIST ---")
    print(dist.to_string(index=False))
    print("\n--- KPI POR DESTINO ---")
    print(d.to_string(index=False))
    print("\n--- EXAMPLES 10x2 ---")
    print(ex.to_string(index=False))


if __name__ == "__main__":
    main()

----------------------------------------------------
[2/65] FILE: \src\audit\audit_desp_poscosecha_ml2.py
----------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime

import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"
```

```python
IN_FINAL = GOLD / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

TS = datetime.now().strftime("%Y%m%d_%H%M%S")

OUT_KPI_G = EVAL / f"audit_desp_poscosecha_ml2_kpi_global_{TS}.parquet"
OUT_KPI_D = EVAL / f"audit_desp_poscosecha_ml2_kpi_by_destino_{TS}.parquet"
OUT_DIST = EVAL / f"audit_desp_poscosecha_ml2_ratio_dist_{TS}.parquet"
OUT_EX = EVAL / f"audit_desp_poscosecha_ml2_examples_10x2_{TS}.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def wmae_log_ratio(log_ratio: pd.Series, w: pd.Series) -> float:
    x = pd.to_numeric(log_ratio, errors="coerce")
    ww = pd.to_numeric(w, errors="coerce").fillna(0.0)
    m = x.notna() & (ww > 0)
    if not bool(m.any()):
        return float("nan")
    denom = float(ww[m].sum())
    if denom <= 0:
        return float(np.nanmean(np.abs(x[m].values)))
    return float((np.abs(x[m].values) * ww[m].values).sum() / denom)


def main() -> None:
    print("\n=== ML2 DESP POSCOSECHA AUDIT ===")

    df = read_parquet(IN_FINAL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha_post_pred_used", "destino", "factor_desp_ml1", "factor_desp_final"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Final sin columnas: {sorted(miss)}")

    df["fecha_post_pred_used"] = _to_date(df["fecha_post_pred_used"])
    df["destino"] = _canon_str(df["destino"])
    df["factor_desp_ml1"] = pd.to_numeric(df["factor_desp_ml1"], errors="coerce")
    df["factor_desp_final"] = pd.to_numeric(df["factor_desp_final"], errors="coerce")

    if "tallos_w" in df.columns:
        w = pd.to_numeric(df["tallos_w"], errors="coerce")
    elif "tallos" in df.columns:
        w = pd.to_numeric(df["tallos"], errors="coerce")
    else:
        w = pd.Series(1.0, index=df.index)

    df["w"] = w.fillna(0.0)


    real = read_parquet(IN_REAL).copy()
    real.columns = [str(c).strip() for c in real.columns]

    need_r = {"fecha_post", "destino", "factor_desp"}
    miss_r = need_r - set(real.columns)
    if miss_r:
        raise ValueError(f"Real sin columnas: {sorted(miss_r)}")

    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    real["factor_desp_real"] = pd.to_numeric(real["factor_desp"], errors="coerce")

    real2 = (
        real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
        .agg(factor_desp_real=("factor_desp_real", "median"))
    )

    x = df.merge(
        real2,
        left_on=["fecha_post_pred_used", "destino"],
        right_on=["fecha_post", "destino"],
        how="left",
    )

    eps = 1e-12
    x["ratio_ml1"] = (x["factor_desp_real"] + eps) / (x["factor_desp_ml1"] + eps)
    x["ratio_ml2"] = (x["factor_desp_real"] + eps) / (x["factor_desp_final"] + eps)
    x["log_ratio_ml1"] = np.log(x["ratio_ml1"])
    x["log_ratio_ml2"] = np.log(x["ratio_ml2"])

    m = x["factor_desp_real"].notna() & x["factor_desp_ml1"].notna() & x["factor_desp_final"].notna()
    d = x.loc[m].copy()

    out_g = pd.DataFrame(
        [
            {
                "n_rows": int(len(d)),
                "n_dates": int(d["fecha_post_pred_used"].nunique()),
                "mae_log_ml1_w": wmae_log_ratio(d["log_ratio_ml1"], d["w"]),
                "mae_log_ml2_w": wmae_log_ratio(d["log_ratio_ml2"], d["w"]),
                "median_ratio_ml1": float(np.nanmedian(d["ratio_ml1"].values)),
                "median_ratio_ml2": float(np.nanmedian(d["ratio_ml2"].values)),
                "improvement_abs_mae_log_w": wmae_log_ratio(d["log_ratio_ml1"], d["w"]) - wmae_log_ratio(d["lo
g_ratio_ml2"], d["w"]),
                "created_at": pd.Timestamp(datetime.now()).normalize(),
            }
        ]
    )

    rows = []
    for dest, g in d.groupby("destino"):
        rows.append(
            {
                "destino": str(dest),
                "n_rows": int(len(g)),
                "n_dates": int(g["fecha_post_pred_used"].nunique()),
                "mae_log_ml1_w": wmae_log_ratio(g["log_ratio_ml1"], g["w"]),
                "mae_log_ml2_w": wmae_log_ratio(g["log_ratio_ml2"], g["w"]),
                "median_ratio_ml2": float(np.nanmedian(g["ratio_ml2"].values)),
                "improvement_abs_mae_log_w": wmae_log_ratio(g["log_ratio_ml1"], g["w"]) - wmae_log_ratio(g["lo
g_ratio_ml2"], g["w"]),
            }
        )
    out_d = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

    dist = pd.DataFrame(
        [
            {
                "n": int(len(d)),
                "ratio_min": float(np.nanmin(d["ratio_ml2"].values)),
                "ratio_p05": float(np.nanpercentile(d["ratio_ml2"].values, 5)),
                "ratio_p25": float(np.nanpercentile(d["ratio_ml2"].values, 25)),
                "ratio_median": float(np.nanmedian(d["ratio_ml2"].values)),
                "ratio_p75": float(np.nanpercentile(d["ratio_ml2"].values, 75)),
                "ratio_p95": float(np.nanpercentile(d["ratio_ml2"].values, 95)),
                "ratio_max": float(np.nanmax(d["ratio_ml2"].values)),
                "created_at": pd.Timestamp(datetime.now()).normalize(),
            }
        ]
    )

    # Examples: score = abs(log_ml1) - abs(log_ml2), ponderado por w
    d["score"] = d["log_ratio_ml1"].abs() - d["log_ratio_ml2"].abs()
    d["wscore"] = d["score"] * d["w"].fillna(0.0)

    best = d.sort_values("wscore", ascending=False).head(10).copy()
    worst = d.sort_values("wscore", ascending=True).head(10).copy()

    best["sample_group"] = "TOP_IMPROVE_10"
    worst["sample_group"] = "TOP_WORSE_10"

    ex = pd.concat([best, worst], ignore_index=True)
    ex_out = ex[
        [
            "fecha",
            "fecha_post_pred_used",
            "destino",
            "w",
            "factor_desp_ml1",
            "factor_desp_final",
            "factor_desp_real",
            "ratio_ml1",
            "ratio_ml2",
            "log_ratio_ml1",
            "log_ratio_ml2",
            "score",
            "wscore",
            "sample_group",
        ]
    ].copy()

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(out_g, OUT_KPI_G)
    write_parquet(out_d, OUT_KPI_D)
    write_parquet(dist, OUT_DIST)
    write_parquet(ex_out, OUT_EX)

    print(f"KPI global parquet : {OUT_KPI_G}")
    print(f"KPI por destino    : {OUT_KPI_D}")
    print(f"Ratio dist parquet : {OUT_DIST}")
    print(f"Examples 10x2      : {OUT_EX}")

    print("\n--- KPI GLOBAL ---")
    print(out_g.to_string(index=False))
    print("\n--- RATIO DIST ---")
    print(dist.to_string(index=False))
    print("\n--- KPI POR DESTINO ---")
    print(out_d.to_string(index=False))
    print("\n--- EXAMPLES 10x2 ---")
    print(ex_out.to_string(index=False))


if __name__ == "__main__":
    main()

--------------------------------------------------
[3/65] FILE: \src\audit\audit_dh_poscosecha_ml2.py
--------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime

import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"

IN_GLOBAL = EVAL / "ml2_dh_poscosecha_eval_global.parquet"
```

```
IN_BY_DEST = EVAL / "ml2_dh_poscosecha_eval_by_destino.parquet"
IN_BY_GR = EVAL / "ml2_dh_poscosecha_eval_by_grado.parquet"
IN_DIST = EVAL / "ml2_dh_poscosecha_eval_delta_dist.parquet"

# Para ejemplos: usamos el backtest factor + KPIs recomputados localmente
IN_FACTOR = EVAL / "backtest_factor_ml2_dh_poscosecha.parquet"
IN_REAL = ROOT / "data" / "silver" / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_KPI_GLOBAL = EVAL / f"audit_dh_poscosecha_ml2_kpi_global_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parque
t"
OUT_KPI_BY_DEST = EVAL / f"audit_dh_poscosecha_ml2_kpi_by_destino_{datetime.now().strftime('%Y%m%d_%H%M%S')}.p
arquet"
OUT_KPI_BY_GR = EVAL / f"audit_dh_poscosecha_ml2_kpi_by_grado_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parqu
et"
OUT_DIST = EVAL / f"audit_dh_poscosecha_ml2_delta_dist_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parquet"
OUT_EX = EVAL / f"audit_dh_poscosecha_ml2_examples_10x2_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def main() -> None:
    print("\n=== ML2 DH POSCOSECHA AUDIT ===")

    g = read_parquet(IN_GLOBAL).copy()
    d = read_parquet(IN_BY_DEST).copy()
    gr = read_parquet(IN_BY_GR).copy()
    dist = read_parquet(IN_DIST).copy()

    # Examples
    fac = read_parquet(IN_FACTOR).copy()
    real = read_parquet(IN_REAL).copy()
    fac.columns = [str(c).strip() for c in fac.columns]
    real.columns = [str(c).strip() for c in real.columns]

    fac["fecha"] = _to_date(fac["fecha"])
    fac["destino"] = _canon_str(fac["destino"])
    fac["grado"] = _canon_int(fac["grado"])
    fac["dh_ml1"] = _canon_int(fac.get("dh_ml1"))
    fac["dh_dias_final"] = _canon_int(fac.get("dh_dias_final"))

    real["fecha_cosecha"] = _to_date(real["fecha_cosecha"])
    real["destino"] = _canon_str(real["destino"])
    real["grado"] = _canon_int(real["grado"])
    real["tallos"] = pd.to_numeric(real.get("tallos"), errors="coerce").fillna(0.0)
    real["dh_dias"] = _canon_int(real.get("dh_dias"))

    real_g = (
        real.groupby(["fecha_cosecha", "grado", "destino"], dropna=False, as_index=False)
            .agg(dh_real=("dh_dias", "median"), tallos=("tallos", "sum"))
    )

    ex = fac.merge(real_g, left_on=["fecha", "grado", "destino"], right_on=["fecha_cosecha", "grado", "destino
"], how="left")
    ex = ex[ex["dh_real"].notna() & ex["dh_ml1"].notna() & ex["dh_dias_final"].notna()].copy()

    ex["err_ml1_days"] = (ex["dh_real"].astype(float) - ex["dh_ml1"].astype(float))
    ex["err_ml2_days"] = (ex["dh_real"].astype(float) - ex["dh_dias_final"].astype(float))
    ex["abs_err_ml1"] = ex["err_ml1_days"].abs()
    ex["abs_err_ml2"] = ex["err_ml2_days"].abs()
    ex["improvement_abs"] = ex["abs_err_ml1"] - ex["abs_err_ml2"]

    # filtrar tallos>0 para evitar ?TOP_BEST? falsos
    ex["tallos"] = pd.to_numeric(ex.get("tallos"), errors="coerce").fillna(0.0)
    ex = ex[ex["tallos"] > 0].copy()

    # Score ponderado por impacto
    ex["wscore"] = ex["improvement_abs"] * ex["tallos"]

    top = ex.sort_values("wscore", ascending=False).head(10).copy()
    top["sample_group"] = "TOP_IMPROVE_10"
    worst = ex.sort_values("wscore", ascending=True).head(10).copy()
    worst["sample_group"] = "TOP_WORSE_10"
    examples = pd.concat([top, worst], ignore_index=True)

    # Outputs
    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(g, OUT_KPI_GLOBAL)
    write_parquet(d, OUT_KPI_BY_DEST)
    write_parquet(gr, OUT_KPI_BY_GR)
    write_parquet(dist, OUT_DIST)
    write_parquet(examples, OUT_EX)

    print(f"KPI global parquet : {OUT_KPI_GLOBAL}")
    print(f"KPI por destino    : {OUT_KPI_BY_DEST}")
    print(f"KPI por grado      : {OUT_KPI_BY_GR}")
    print(f"Delta dist parquet : {OUT_DIST}")
    print(f"Examples 10x2      : {OUT_EX}")

    print("\n--- KPI GLOBAL ---")
    print(g.to_string(index=False))
    print("\n--- DELTA DIST ---")
    print(dist.to_string(index=False))
    print("\n--- KPI DESTINO ---")
    print(d.to_string(index=False))
    print("\n--- KPI GRADO (head) ---")
    print(gr.head(10).to_string(index=False))
    print("\n--- EXAMPLES (10x2) ---")
    keep = [c for c in ["fecha", "grado", "destino", "tallos", "dh_ml1", "dh_dias_final", "dh_real", "err_ml1_
days", "err_ml2_days", "improvement_abs", "wscore", "sample_group"] if c in examples.columns]
    print(examples[keep].to_string(index=False))


if __name__ == "__main__":
    main()

----------------------------------------------------
[4/65] FILE: \src\audit\audit_harvest_horizon_ml2.py
----------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_FACTOR = EVAL_DIR / "backtest_factor_ml2_harvest_horizon.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def _mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else float("nan")


def _bias(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(x)) if len(x) else float("nan")


def _pick_first_existing(df: pd.DataFrame, candidates: list[str]) -> str:
    for c in candidates:
        if c in df.columns:
            return c
    raise KeyError(f"None of these columns exist: {candidates}")


def main() -> None:
    ciclo = read_parquet(IN_CICLO).copy()
    fact = read_parquet(IN_FACTOR).copy()

    # Normalizar fechas en ciclo (autoridad de lo real)
    # Nota: ciclo puede tener nombres distintos según tu versión, cubrimos ambos.
    c_ini = _pick_first_existing(ciclo, ["fecha_inicio_cosecha", "fecha_ini_cosecha"])
    c_fin = _pick_first_existing(ciclo, ["fecha_fin_cosecha", "fecha_fin"])

    ciclo[c_ini] = _to_date(ciclo[c_ini])
    ciclo[c_fin] = _to_date(ciclo[c_fin])

    # Para evitar sufijos _x/_y: tomamos de ciclo solo lo que necesitamos
    cols = ["ciclo_id", c_ini, c_fin]
    if "estado" in ciclo.columns:
        cols.insert(1, "estado")
    ciclo_small = ciclo[cols].copy()

    ciclo_small = ciclo_small.rename(columns={c_ini: "fecha_inicio_cosecha_real", c_fin: "fecha_fin_cosecha_re
al"})

    df = ciclo_small.merge(fact, on="ciclo_id", how="inner")
    # Estado: puede venir en ciclo o en factor. Normalizamos y garantizamos que exista.
    if "estado" not in df.columns:
        if "estado_x" in df.columns:
            df["estado"] = df["estado_x"]
        elif "estado_y" in df.columns:
            df["estado"] = df["estado_y"]
        elif "estado_ml1" in df.columns:
            df["estado"] = df["estado_ml1"]
        else:
            df["estado"] = "UNKNOWN"

    df["estado"] = df["estado"].astype(str).str.upper().str.strip()

    # Real duration
    df["n_harvest_days_real"] = (df["fecha_fin_cosecha_real"] - df["fecha_inicio_cosecha_real"]).dt.days + 1

    # Errors (duración)
    df["err_ml1_days"] = pd.to_numeric(df["n_harvest_days_real"], errors="coerce") - pd.to_numeric(df["n_harve
st_days_pred"], errors="coerce")
    df["err_ml2_days"] = pd.to_numeric(df["n_harvest_days_real"], errors="coerce") - pd.to_numeric(df["n_harve
st_days_final"], errors="coerce")

    df = df.loc[df["err_ml1_days"].notna() & df["err_ml2_days"].notna(), :].copy()

    # Paths outputs con timestamp
    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    out_kpi_global = EVAL_DIR / f"audit_harvest_horizon_ml2_kpi_global_{ts}.parquet"
    out_dist = EVAL_DIR / f"audit_harvest_horizon_ml2_adjust_dist_{ts}.parquet"
    out_by_estado = EVAL_DIR / f"audit_harvest_horizon_ml2_kpi_by_estado_{ts}.parquet"
    out_examples = EVAL_DIR / f"audit_harvest_horizon_ml2_examples_10x2_{ts}.parquet"

    # KPI GLOBAL
```

```
kpi_global = pd.DataFrame([{
    "n": int(len(df)),
    "mae_ml1_days": _mae(df["err_ml1_days"]),
    "mae_ml2_days": _mae(df["err_ml2_days"]),
    "bias_ml1_days": _bias(df["err_ml1_days"]),
    "bias_ml2_days": _bias(df["err_ml2_days"]),
    "improvement_abs_days": (_mae(df["err_ml1_days"]) - _mae(df["err_ml2_days"])),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
}])
write_parquet(kpi_global, out_kpi_global)

# DIST AJUSTE
adj = pd.to_numeric(df["pred_error_horizon_days"], errors="coerce")
dist = pd.DataFrame([{
    "n_factor_rows": int(adj.notna().sum()),
    "adj_min": float(np.nanmin(adj)) if len(adj) else np.nan,
    "adj_p25": float(np.nanpercentile(adj, 25)) if len(adj) else np.nan,
    "adj_median": float(np.nanmedian(adj)) if len(adj) else np.nan,
    "adj_p75": float(np.nanpercentile(adj, 75)) if len(adj) else np.nan,
    "adj_max": float(np.nanmax(adj)) if len(adj) else np.nan,
    "pct_clip": float(np.nanmean((adj <= -14) | (adj >= 21))) if len(adj) else np.nan,
}])
write_parquet(dist, out_dist)

# KPI POR ESTADO
rows = []
for estado, g in df.groupby("estado"):
    rows.append({
        "estado": estado,
        "n": int(len(g)),
        "mae_ml1_days": _mae(g["err_ml1_days"]),
        "mae_ml2_days": _mae(g["err_ml2_days"]),
        "bias_ml1_days": _bias(g["err_ml1_days"]),
        "bias_ml2_days": _bias(g["err_ml2_days"]),
        "improvement_abs_days": (_mae(g["err_ml1_days"]) - _mae(g["err_ml2_days"])),
    })
by_estado = pd.DataFrame(rows).sort_values(["estado"])
write_parquet(by_estado, out_by_estado)

# EJEMPLOS 10x2: 10 CERRADO + 10 ACTIVO (si existen), priorizando mayor |err_ml1|
df["abs_err_ml1"] = pd.to_numeric(df["err_ml1_days"], errors="coerce").abs()
examples = (
    df.sort_values("abs_err_ml1", ascending=False)
      .groupby("estado", group_keys=False)
      .head(10)
      .copy()
)
write_parquet(examples, out_examples)

print("\n=== ML2 HARVEST HORIZON AUDIT ===")
print(f"KPI global parquet : {out_kpi_global}")
print(f"Dist ajustes parquet: {out_dist}")
print(f"KPI por estado      : {out_by_estado}")
print(f"Examples 10x2       : {out_examples}")

print("\n--- KPI GLOBAL ---")
print(kpi_global.to_string(index=False))
print("\n--- AJUSTE DIST ---")
print(dist.to_string(index=False))
print("\n--- KPI POR ESTADO ---")
print(by_estado.to_string(index=False))
print("\n--- EXAMPLES (head) ---")
cols_show = [c for c in [
    "ciclo_id", "estado",
    "fecha_inicio_cosecha_real", "fecha_fin_cosecha_real",
    "n_harvest_days_pred", "pred_error_horizon_days", "n_harvest_days_final",
    "err_ml1_days", "err_ml2_days",
    "ml1_version", "ml2_run_id",
] if c in examples.columns]
print(examples[cols_show].head(10).to_string(index=False))
```

```
if __name__ == "__main__":
    main()

--------------------------------------------------
[5/65] FILE: \src\audit\audit_harvest_start_ml2.py
--------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    # .../src/eval/file.py -> repo_root = parents[2]
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_FACTOR = GOLD_DIR / "factors" / "factor_ml2_harvest_start.parquet"
IN_DS = GOLD_DIR / "ml2_datasets" / "ds_harvest_start_ml2_v2.parquet"  # para features clima agregadas


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else float("nan")


def _bias(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(x)) if len(x) else float("nan")


def main() -> None:
    # ---------- Load ----------
    ciclo = read_parquet(IN_CICLO).copy()
    fact = read_parquet(IN_FACTOR).copy()

    # Dataset v2 (para features clima as_of); puede no existir en prod, así que lo protegemos
    ds = None
    if IN_DS.exists():
        ds = read_parquet(IN_DS).copy()

    # ---------- Canon / dates ----------
    ciclo["estado"] = _canon_str(ciclo["estado"])
    ciclo["bloque_base"] = _canon_str(ciclo["bloque_base"])
    ciclo["fecha_sp"] = _to_date(ciclo["fecha_sp"])
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])

    fact["bloque_base"] = _canon_str(fact["bloque_base"])
    if "estado" in fact.columns:
        fact["estado"] = _canon_str(fact["estado"])

    fact["fecha_sp"] = _to_date(fact["fecha_sp"])
    fact["harvest_start_pred"] = _to_date(fact["harvest_start_pred"])
    fact["harvest_start_final"] = _to_date(fact["harvest_start_final"])

    # ---------- Join core audit frame ----------
    df = ciclo.merge(
        fact,
        on="ciclo_id",
        how="inner",
        suffixes=("_real", "_ml2"),
    )

    # usar estado de ciclo maestro como autoridad
    df["estado"] = df["estado_real"]

    # ---------- Errors ----------
    df["err_ml1_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_pred"]).dt.days
    df["err_ml2_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_final"]).dt.days

    # Algunas filas pueden no tener real (activos sin inicio real aún). Filtramos para KPI real.
    df_eval = df.loc[df["fecha_inicio_cosecha"].notna() & df["harvest_start_pred"].notna() & df["harvest_start
_final"].notna(), :].copy()

    # ---------- KPI global ----------
    mae_ml1 = _mae(df_eval["err_ml1_days"])
    mae_ml2 = _mae(df_eval["err_ml2_days"])
    bias_ml1 = _bias(df_eval["err_ml1_days"])
    bias_ml2 = _bias(df_eval["err_ml2_days"])

    kpi_global = pd.DataFrame([{
        "n": int(len(df_eval)),
        "mae_ml1_days": mae_ml1,
        "mae_ml2_days": mae_ml2,
        "bias_ml1_days": bias_ml1,
        "bias_ml2_days": bias_ml2,
        "improvement_abs_days": (mae_ml1 - mae_ml2) if (pd.notna(mae_ml1) and pd.notna(mae_ml2)) else np.nan,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    # ---------- Adjustment distribution ----------
    adj = pd.to_numeric(df.get("pred_error_start_days"), errors="coerce")
    dist = pd.DataFrame([{
        "n_factor_rows": int(len(df)),
        "adj_min": float(np.nanmin(adj)) if len(adj) else np.nan,
        "adj_p25": float(np.nanpercentile(adj, 25)) if len(adj) else np.nan,
        "adj_median": float(np.nanmedian(adj)) if len(adj) else np.nan,
        "adj_p75": float(np.nanpercentile(adj, 75)) if len(adj) else np.nan,
        "adj_max": float(np.nanmax(adj)) if len(adj) else np.nan,
    }])

    # % clipping (asumimos guardrail ±21)
    CLIP_LO, CLIP_HI = -21, 21
    clip_mask = (adj <= CLIP_LO) | (adj >= CLIP_HI)
    dist["pct_clip"] = float(np.nanmean(clip_mask)) if len(adj) else np.nan

    # ---------- KPI por estado (ABIERTO/CERRADO) ----------
    by_estado_rows = []
    for est, g in df_eval.groupby("estado"):
        mae1 = _mae(g["err_ml1_days"])
        mae2 = _mae(g["err_ml2_days"])
        by_estado_rows.append({
            "estado": est,
            "n": int(len(g)),
            "mae_ml1_days": mae1,
            "mae_ml2_days": mae2,
            "bias_ml1_days": _bias(g["err_ml1_days"]),
            "bias_ml2_days": _bias(g["err_ml2_days"]),
            "improvement_abs_days": (mae1 - mae2) if (pd.notna(mae1) and pd.notna(mae2)) else np.nan,
        })
    kpi_by_estado = pd.DataFrame(by_estado_rows).sort_values(["estado"])

    # ---------- ?Sentido agronómico? simple (si hay dataset con features) ----------
    # Join con ds (tiene gdc_cum_sp, rain_cum_sp, etc. a nivel ciclo_id+as_of_date).
    # Para audit, tomamos el último as_of por ciclo (más cercano al inicio real).
    agr = pd.DataFrame()
    if ds is not None and len(ds):
        ds = ds.copy()
        ds["as_of_date"] = _to_date(ds["as_of_date"])
        # último as_of por ciclo
        ds_last = ds.sort_values(["ciclo_id", "as_of_date"]).groupby("ciclo_id", as_index=False).tail(1)
        # join
        wanted = ["ciclo_id", "gdc_cum_sp", "gdc_14d", "rain_7d", "rain_cum_sp", "solar_7d", "temp_avg_7d"]
        cols = [c for c in wanted if c in ds_last.columns]
        agr = df.merge(ds_last[cols].copy(), on="ciclo_id", how="left")


        # Correlaciones simples (no causal proof, solo sanity)
        rows = []
        feat_list = [c for c in ["gdc_cum_sp", "gdc_14d", "rain_7d", "rain_cum_sp", "solar_7d", "temp_avg_7d"]
if c in agr.columns]
        for col in feat_list:

            x = pd.to_numeric(agr[col], errors="coerce")
            y = pd.to_numeric(agr["pred_error_start_days"], errors="coerce")
            m = x.notna() & y.notna()
            if m.sum() >= 30:
                corr = float(np.corrcoef(x[m], y[m])[0, 1])
            else:
                corr = np.nan
            rows.append({"feature": col, "corr_with_pred_error_days": corr, "n": int(m.sum())})
        agr_corr = pd.DataFrame(rows)
    else:
        agr_corr = pd.DataFrame([{"feature": None, "corr_with_pred_error_days": np.nan, "n": 0}])

    # ---------- 10 ejemplos ABIERTO y 10 CERRADO ----------
    # Muestra reproducible, y prioriza casos con mayor ajuste absoluto para que sea informativo.
    def _pick_examples(est: str, n: int = 10) -> pd.DataFrame:
        g = df.copy()
        g = g.loc[g["estado"] == est, :].copy()
        g["abs_adj"] = pd.to_numeric(g["pred_error_start_days"], errors="coerce").abs()
        # si no hay suficientes, devuelve lo que haya
        g = g.sort_values(["abs_adj"], ascending=False).head(max(n * 3, n))  # pool
        # sample dentro del pool para variedad
        if len(g) > n:
            g = g.sample(n=n, random_state=42)
        cols = [
            "ciclo_id", "bloque_base_real", "variedad", "area", "tipo_sp",
            "estado",
            "fecha_sp_real", "fecha_inicio_cosecha",
            "harvest_start_pred", "harvest_start_final",
            "pred_error_start_days",
            "err_ml1_days", "err_ml2_days",
            "ml1_version", "ml2_run_id",
        ]
        # Algunas columnas tienen sufijos por el merge; normalizamos nombres:
        rename = {
            "bloque_base_real": "bloque_base",
            "fecha_sp_real": "fecha_sp",
        }
        out = g[[c for c in cols if c in g.columns]].rename(columns=rename)
        return out

    ex_abierto = _pick_examples("ABIERTO", 10)
    ex_cerrado = _pick_examples("CERRADO", 10)

    examples = pd.concat(
        [
            ex_abierto.assign(sample_group="ABIERTO_10"),
            ex_cerrado.assign(sample_group="CERRADO_10"),
        ],
        ignore_index=True,
    )

    # ---------- Save outputs ----------
    EVAL_DIR.mkdir(parents=True, exist_ok=True)
    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    out_kpi_global = EVAL_DIR / f"audit_harvest_start_ml2_kpi_global_{ts}.parquet"
    out_dist = EVAL_DIR / f"audit_harvest_start_ml2_adjust_dist_{ts}.parquet"
    out_by_estado = EVAL_DIR / f"audit_harvest_start_ml2_kpi_by_estado_{ts}.parquet"
    out_corr = EVAL_DIR / f"audit_harvest_start_ml2_agro_corr_{ts}.parquet"
    out_examples = EVAL_DIR / f"audit_harvest_start_ml2_examples_10x2_{ts}.parquet"

    write_parquet(kpi_global, out_kpi_global)
    write_parquet(dist, out_dist)
    write_parquet(kpi_by_estado, out_by_estado)
    write_parquet(agr_corr, out_corr)
    write_parquet(examples, out_examples)

    # CSV opcional (muy útil para revisar rápido)
    examples.to_csv(EVAL_DIR / f"audit_harvest_start_ml2_examples_10x2_{ts}.csv", index=False)

    # ---------- Print summary ----------
    print("\n=== ML2 HARVEST START AUDIT ===")
    print(f"KPI global parquet : {out_kpi_global}")
    print(f"Dist ajustes parquet: {out_dist}")
    print(f"KPI por estado      : {out_by_estado}")
    print(f"Corr agronómica     : {out_corr}")
    print(f"Examples 10x2       : {out_examples}")
    print("\n--- KPI GLOBAL ---")
    print(kpi_global.to_string(index=False))
    print("\n--- AJUSTE DIST ---")
    print(dist.to_string(index=False))
    print("\n--- KPI POR ESTADO ---")
    print(kpi_by_estado.to_string(index=False))
    print("\n--- CORR (sanity) ---")
    print(agr_corr.to_string(index=False))
    print("\n--- EXAMPLES (top) ---")
    print(examples.head(10).to_string(index=False))


if __name__ == "__main__":
    main()

----------------------------------------------------
[6/65] FILE: \src\audit\audit_hidr_poscosecha_ml2.py
----------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"

IN_GLOBAL = EVAL / "ml2_hidr_poscosecha_eval_global.parquet"
IN_BY_DESTINO = EVAL / "ml2_hidr_poscosecha_eval_by_destino.parquet"
IN_BY_GRADO = EVAL / "ml2_hidr_poscosecha_eval_by_grado.parquet"
IN_RATIO_DIST = EVAL / "ml2_hidr_poscosecha_eval_ratio_dist.parquet"

# Para ejemplos necesitamos final + real:
GOLD = DATA / "gold"
SILVER = DATA / "silver"
IN_FINAL = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _factor_from_hidr_pct(hidr_pct: pd.Series) -> pd.Series:
    x = pd.to_numeric(hidr_pct, errors="coerce")
    return np.where(x.isna(), np.nan, np.where(x > 3.5, 1.0 + x / 100.0, x)).astype(float)


def main() -> None:
    ts = datetime.now().strftime("%Y%m%d_%H%M%S")

    out_global = EVAL / f"audit_hidr_poscosecha_ml2_kpi_global_{ts}.parquet"
    out_dest = EVAL / f"audit_hidr_poscosecha_ml2_kpi_by_destino_{ts}.parquet"
    out_gr = EVAL / f"audit_hidr_poscosecha_ml2_kpi_by_grado_{ts}.parquet"
    out_dist = EVAL / f"audit_hidr_poscosecha_ml2_ratio_dist_{ts}.parquet"
    out_examples = EVAL / f"audit_hidr_poscosecha_ml2_examples_10x2_{ts}.parquet"

    g = read_parquet(IN_GLOBAL).copy()
    bd = read_parquet(IN_BY_DESTINO).copy()
    bg = read_parquet(IN_BY_GRADO).copy()
    dist = read_parquet(IN_RATIO_DIST).copy()

    # ejemplos
    fin = read_parquet(IN_FINAL).copy()
    fin.columns = [str(c).strip() for c in fin.columns]

    # fecha_post_pred usada
    fpp = None
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml2", "fecha_post_pred_ml1", "
fecha_post_pred"]:
        if c in fin.columns:
            fpp = c
            break
    if fpp is None:
        raise KeyError("No encuentro fecha_post_pred en final para ejemplos.")

    fin[fpp] = _to_date(fin[fpp])
    fin["destino"] = _canon_str(fin["destino"])
    fin["grado"] = _canon_int(fin["grado"])
    fin["factor_hidr_ml1"] = pd.to_numeric(fin.get("factor_hidr_ml1"), errors="coerce")
    fin["factor_hidr_final"] = pd.to_numeric(fin.get("factor_hidr_final"), errors="coerce")

    real = read_parquet(IN_REAL).copy()
    real.columns = [str(c).strip() for c in real.columns]
    if "hidr_pct" in real.columns:
        real["factor_hidr_real"] = _factor_from_hidr_pct(real["hidr_pct"])
    else:
        pb = pd.to_numeric(real.get("peso_base_g"), errors="coerce")
        pp = pd.to_numeric(real.get("peso_post_g"), errors="coerce")
        real["factor_hidr_real"] = np.where(pb > 0, pp / pb, np.nan)

    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    real["grado"] = _canon_int(real["grado"])

    if "tallos" in real.columns:
        real["tallos"] = pd.to_numeric(real["tallos"], errors="coerce").fillna(0.0)
        g2 = real.groupby(["fecha_post", "grado", "destino"], dropna=False)
        real2 = g2.apply(
            lambda x: pd.Series({
                "factor_hidr_real": float(np.nansum(x["factor_hidr_real"] * x["tallos"]) / np.nansum(x["tallos
"])) if np.nansum(x["tallos"]) > 0 else float(np.nanmedian(x["factor_hidr_real"])),
                "tallos": float(np.nansum(x["tallos"])),
            })
        ).reset_index()
    else:
        real2 = (
            real.groupby(["fecha_post", "grado", "destino"], dropna=False, as_index=False)
                .agg(factor_hidr_real=("factor_hidr_real", "median"))
        )
        real2["tallos"] = 1.0

    df = fin.merge(
        real2,
        left_on=[fpp, "grado", "destino"],
        right_on=["fecha_post", "grado", "destino"],
        how="left",
    ).drop(columns=["fecha_post"], errors="ignore")

    m = df["factor_hidr_real"].notna() & df["factor_hidr_ml1"].notna() & df["factor_hidr_final"].notna()
    d = df.loc[m].copy()
    eps = 1e-9
    d["ratio_ml1"] = d["factor_hidr_real"] / d["factor_hidr_ml1"].clip(lower=eps)
    d["ratio_ml2"] = d["factor_hidr_real"] / d["factor_hidr_final"].clip(lower=eps)
    d["abs_log_ml1"] = np.abs(np.log(d["ratio_ml1"].clip(lower=eps)))
    d["abs_log_ml2"] = np.abs(np.log(d["ratio_ml2"].clip(lower=eps)))

    # score ponderado por tallos (impacto)
    w = pd.to_numeric(d.get("tallos"), errors="coerce").fillna(1.0).clip(lower=0.0)
    d["wscore"] = d["abs_log_ml2"] * w

    # top improve/worst según mejora en abs_log (ponderado)
    d["impr"] = (d["abs_log_ml1"] - d["abs_log_ml2"]) * w

    best = d.sort_values("impr", ascending=False).head(10).assign(sample_group="TOP_IMPROVE_10")
    worst = d.sort_values("impr", ascending=True).head(10).assign(sample_group="TOP_WORSE_10")
    ex = pd.concat([best, worst], ignore_index=True)

    ex_out = ex[[
        "fecha", fpp, "grado", "destino", "factor_hidr_ml1", "factor_hidr_final",
        "factor_hidr_real", "ratio_ml1", "ratio_ml2", "impr", "wscore", "sample_group"
    ]].rename(columns={fpp: "fecha_post_pred_used"})

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(g, out_global)
    write_parquet(bd, out_dest)
    write_parquet(bg, out_gr)
    write_parquet(dist, out_dist)
    write_parquet(ex_out, out_examples)

    print("\n=== ML2 HIDR POSCOSECHA AUDIT ===")
    print(f"KPI global parquet : {out_global}")
    print(f"KPI por destino    : {out_dest}")
    print(f"KPI por grado      : {out_gr}")
    print(f"Ratio dist parquet : {out_dist}")
    print(f"Examples 10x2      : {out_examples}")

    print("\n--- KPI GLOBAL ---")
    print(g.to_string(index=False))
    print("\n--- RATIO DIST ---")
    print(dist.to_string(index=False))
    print("\n--- KPI DESTINO ---")
    print(bd.to_string(index=False))
    print("\n--- KPI GRADO (head) ---")
    print(bg.head(13).to_string(index=False))
    print("\n--- EXAMPLES (10x2) ---")
    print(ex_out.to_string(index=False))


if __name__ == "__main__":
    main()

--------------------------------------------------------
[7/65] FILE: \src\audit\audit_ml1_curva_share_vs_real.py
--------------------------------------------------------
from __future__ import annotations
```

```python
from pathlib import Path
import json
import numpy as np
import pandas as pd

from common.io import read_parquet


# -------------------------
# Paths
# -------------------------
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")

# Baseline día (para comparar forma)
OFERTA_PATH = Path("data/preds/pred_oferta_dia.parquet")

# Predicciones nuevas por share (SALIDA del apply_curva_share_dia)
# OJO: si tu apply escribe en pred_factor_curva_ml1.parquet, úsalo.
PRED_FACTOR_PATH = Path("data/gold/pred_factor_curva_ml1.parquet")

# Para reconstruir tallos_ml1_dia: baseline * factor (compat downstream)
OUT_REPORT = Path("data/audit/audit_ml1_curva_share_report.parquet")


# -------------------------
# Helpers
# -------------------------
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Cols={list(df.columns)}")


def _safe_div(a: np.ndarray, b: np.ndarray, eps: float = 1e-12) -> np.ndarray:
    return a / (b + eps)


def _cycle_metrics(df: pd.DataFrame) -> pd.Series:
    """
    df: rows de un ciclo con columnas:
      - tallos_real_dia (>=0)
      - tallos_ml1_dia (>=0)
      - tallos_base_dia (>=0)
    """
    r = df["tallos_real_dia"].to_numpy(dtype=float)
    m = df["tallos_ml1_dia"].to_numpy(dtype=float)

    sr = r.sum()
    sm = m.sum()

    if sr <= 0:
        return pd.Series(
            {
                "has_real": False,
                "l1_share": np.nan,
                "ks_cdf": np.nan,
                "peak_pos_err_days": np.nan,
                "mass_early_diff": np.nan,
                "mass_tail_diff": np.nan,
                "n_days": int(len(df)),
            }
        )

    # shares
    pr = r / sr
    pm = np.where(sm > 0, m / sm, 0.0)

    # L1 share distance
    l1 = float(np.abs(pm - pr).sum())

    # KS on CDF
    cdf_r = np.cumsum(pr)
    cdf_m = np.cumsum(pm)
    ks = float(np.max(np.abs(cdf_m - cdf_r)))

    # Peak position error (argmax day index)
    peak_r = int(np.argmax(pr))
    peak_m = int(np.argmax(pm))
    peak_err = peak_m - peak_r

    # Early/tail mass diffs (primer 20% y último 20%)
    n = len(df)
    k = max(1, int(np.ceil(0.20 * n)))
    early_r = float(pr[:k].sum())
    early_m = float(pm[:k].sum())
    tail_r = float(pr[-k:].sum())
    tail_m = float(pm[-k:].sum())

    return pd.Series(
        {
            "has_real": True,
            "l1_share": l1,
            "ks_cdf": ks,
            "peak_pos_err_days": peak_err,
            "mass_early_diff": early_m - early_r,
            "mass_tail_diff": tail_m - tail_r,
            "n_days": int(n),
        }
    )


def main() -> None:
    created_at = pd.Timestamp.utcnow()

    # -------------------------
    # Read
    # -------------------------
    uni = read_parquet(UNIVERSE_PATH).copy()
    prog = read_parquet(PROG_PATH).copy()
    oferta = read_parquet(OFERTA_PATH).copy()
    pred = read_parquet(PRED_FACTOR_PATH).copy()

    # -------------------------
    # Canon keys
    # -------------------------
    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe")
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key].drop_duplicates()

    # PROG: viene variedad raw => canonizar mínimo (XLENCE->XL, CLOUD->CLO).
    _require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "prog")
    prog["ciclo_id"] = prog["ciclo_id"].astype(str)
    prog["fecha"] = _to_date(prog["fecha"])
    prog["bloque_base"] = _canon_int(prog["bloque_base"])
    prog["variedad_raw"] = _canon_str(prog["variedad"])
    prog["variedad_canon"] = prog["variedad_raw"].replace({"XLENCE": "XL", "CLOUD": "CLO"})
    prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0).astype(float
)
    prog_k = prog[["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_real_dia"]].drop_duplicates(s
ubset=key)

    # OFERTA baseline
    _require(oferta, ["ciclo_id", "fecha", "bloque_base", "stage", "tallos_pred"], "oferta")
    oferta["ciclo_id"] = oferta["ciclo_id"].astype(str)
    oferta["fecha"] = _to_date(oferta["fecha"])
    oferta["bloque_base"] = _canon_int(oferta["bloque_base"])
    oferta["stage"] = _canon_str(oferta["stage"])
    # canon variedad (si ya existe en oferta, ok; si no, fallback de tu pipeline)
    if "variedad_canon" in oferta.columns:
        oferta["variedad_canon"] = _canon_str(oferta["variedad_canon"])
    elif "variedad" in oferta.columns:
        oferta["variedad_canon"] = _canon_str(oferta["variedad"]).replace({"XLENCE": "XL", "CLOUD": "CLO"})
    else:
        oferta["variedad_canon"] = "UNKNOWN"
    oferta = oferta[oferta["stage"].eq("HARVEST")].copy()
    oferta["tallos_base_dia"] = pd.to_numeric(oferta["tallos_pred"], errors="coerce").fillna(0.0).astype(float
)
    oferta_k = (
        oferta.groupby(key, as_index=False)
        .agg(tallos_base_dia=("tallos_base_dia", "sum"),
             tallos_proy=("tallos_proy", "max") if "tallos_proy" in oferta.columns else ("tallos_base_dia", "s
um"))
    )

    # PRED factor / share
    _require(pred, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "factor_curva_ml1"], "pred_factor")
    pred["ciclo_id"] = pred["ciclo_id"].astype(str)
    pred["fecha"] = _to_date(pred["fecha"])
    pred["bloque_base"] = _canon_int(pred["bloque_base"])
    pred["variedad_canon"] = _canon_str(pred["variedad_canon"])
    pred["factor_curva_ml1"] = pd.to_numeric(pred["factor_curva_ml1"], errors="coerce").fillna(1.0).astype(flo
at)

    pred_k = pred[key + ["factor_curva_ml1"]].drop_duplicates(subset=key)

    # -------------------------
    # Panel
    # -------------------------
    panel = (
        uni_k.merge(oferta_k, on=key, how="left")
             .merge(pred_k, on=key, how="left")
             .merge(prog_k, on=key, how="left")
    )

    panel["tallos_base_dia"] = pd.to_numeric(panel["tallos_base_dia"], errors="coerce").fillna(0.0).astype(flo
at)
    panel["factor_curva_ml1"] = pd.to_numeric(panel["factor_curva_ml1"], errors="coerce").fillna(1.0).astype(f
loat)
    grp = ["ciclo_id"]
    panel["_f"] = panel["factor_curva_ml1"].clip(lower=0.0)

    s = panel.groupby(grp, dropna=False)["_f"].transform("sum")
    panel["_w"] = np.where(s > 0, panel["_f"] / s, 0.0)

    panel["tallos_ml1_dia"] = panel["_w"] * panel["tallos_proy"]

    panel["tallos_real_dia"] = pd.to_numeric(panel["tallos_real_dia"], errors="coerce").fillna(0.0).astype(flo
at)

    # Coverage
    universe_rows = int(len(uni_k))
    miss_oferta = int(panel["tallos_base_dia"].isna().sum())  # (debería ser 0 tras fillna, pero dejamos)
    miss_pred = int(panel["factor_curva_ml1"].isna().sum())
    cov_real = float((panel["tallos_real_dia"] > 0).mean())

    print("=== COVERAGE ===")
    print(
        pd.DataFrame([{
            "created_at": created_at,
            "universe_rows": universe_rows,
            "coverage_real_gt0": cov_real,
            "miss_oferta_rows": miss_oferta,
            "miss_pred_rows": miss_pred,
        }]).to_string(index=False)
    )

    # -------------------------
    # Invariants
    # -------------------------
    # Mass balance ML1 vs proy (si tallos_proy existe y es consistente)
    if "tallos_proy" in panel.columns:
        cyc = panel.groupby("ciclo_id", dropna=False).agg(
            proy=("tallos_proy", "max"),
            ml1_sum=("tallos_ml1_dia", "sum"),
        )
        cyc["abs_diff"] = (cyc["proy"].astype(float) - cyc["ml1_sum"].astype(float)).abs()
        print("\n=== MASS BALANCE (cycle) ===")
        print(f"cycles={len(cyc):,} | max abs diff ml1 vs proy: {float(cyc['abs_diff'].max()):.12f}")

    # -------------------------
    # Shape metrics (cycle-level) for cycles with real>0
    # -------------------------
    # Orden por fecha dentro de ciclo
    panel = panel.sort_values(["ciclo_id", "fecha"]).reset_index(drop=True)

    cyc_metrics = panel.groupby("ciclo_id", dropna=False).apply(_cycle_metrics).reset_index()

    n_total = int(cyc_metrics["ciclo_id"].nunique())
    n_real = int(cyc_metrics["has_real"].sum())
    print("\n=== SHAPE (cycle) ===")
    print(f"cycles total={n_total:,} | cycles with real={n_real:,}")

    if n_real > 0:
        cols_show = ["l1_share", "ks_cdf", "peak_pos_err_days", "mass_early_diff", "mass_tail_diff"]
        for c in cols_show:
            s = cyc_metrics.loc[cyc_metrics["has_real"], c].astype(float)
            print(f"\n{c}:")
            print(s.describe().to_string())

    # -------------------------
    # Save detailed report
    # -------------------------
    panel["created_at_audit"] = created_at
    OUT_REPORT.parent.mkdir(parents=True, exist_ok=True)
    write = panel  # full panel for drill-down
    # guardamos también métricas por ciclo en JSON sidecar
    cyc_out = cyc_metrics.copy()
    cyc_out["created_at_audit"] = created_at
    write.to_parquet(OUT_REPORT, index=False)

    cyc_path = OUT_REPORT.with_name("audit_ml1_curva_share_cycle_metrics.parquet")
    cyc_out.to_parquet(cyc_path, index=False)

    print(f"\nOK -> {OUT_REPORT}")
    print(f"OK -> {cyc_path}")


if __name__ == "__main__":
    main()

-------------------------------------------------
[8/65] FILE: \src\audit\audit_ml1_curva_shares.py
-------------------------------------------------
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd
```

```python
from common.io import read_parquet, write_parquet


# =========================
# Paths
# =========================
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")
PRED_FULL_PATH = Path("data/gold/pred_tallos_grado_dia_ml1_full.parquet")

CURVA_PATH = Path("data/gold/pred_factor_curva_ml1.parquet")
DIST_PATH = Path("data/gold/pred_dist_grado_ml1.parquet")
OFERTA_PATH = Path("data/preds/pred_oferta_dia.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")

DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")

AUDIT_DIR = Path("data/audit/ml1_status")


# =========================
# Helpers
# =========================
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")


def _load_var_map() -> dict[str, str]:
    if not DIM_VAR_PATH.exists():
        # fallback duro (tu convención)
        return {"XLENCE": "XL", "XL": "XL", "CLOUD": "CLO", "CLO": "CLO"}
    dv = read_parquet(DIM_VAR_PATH).copy()
    _require(dv, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv["raw"] = _canon_str(dv["variedad_raw"])
    dv["canon"] = _canon_str(dv["variedad_canon"])
    m = dict(zip(dv["raw"], dv["canon"]))
    # asegurar mínimos
    m.setdefault("XLENCE", "XL")
    m.setdefault("XL", "XL")
    m.setdefault("CLOUD", "CLO")
    m.setdefault("CLO", "CLO")
    return m


def _canon_var_from_col(df: pd.DataFrame, col: str, var_map: dict[str, str]) -> pd.Series:
    raw = _canon_str(df[col]) if col in df.columns else pd.Series(["UNKNOWN"] * len(df))
    return raw.map(var_map).fillna(raw)


def _ensure_relpos(uni: pd.DataFrame) -> pd.DataFrame:
    out = uni.copy()
    if "rel_pos" not in out.columns and "rel_pos_pred" in out.columns:
        out["rel_pos"] = pd.to_numeric(out["rel_pos_pred"], errors="coerce")
    if "day_in_harvest" not in out.columns and "day_in_harvest_pred" in out.columns:
        out["day_in_harvest"] = pd.to_numeric(out["day_in_harvest_pred"], errors="coerce").astype("Int64")
    if "n_harvest_days" not in out.columns and "n_harvest_days_pred" in out.columns:
        out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days_pred"], errors="coerce").astype("Int64")
    return out
```

```
def _cycle_shape_metrics(sub: pd.DataFrame) -> dict:
    # requiere: fecha, share_ml1, share_real opcional, rel_pos opcional
    sub = sub.sort_values("fecha").copy()

    ml1 = pd.to_numeric(sub["share_ml1"], errors="coerce").fillna(0.0).to_numpy(float)
    ml1 = np.clip(ml1, 0.0, None)
    ml1 = ml1 / (ml1.sum() + 1e-12)

    out = {
        "n_days": int(len(sub)),
        "has_real": 0,
        "peak_idx_ml1": int(np.argmax(ml1)) if len(ml1) else pd.NA,
        "peak_share_ml1": float(np.max(ml1)) if len(ml1) else pd.NA,
        "l1_share": pd.NA,
        "ks_cdf": pd.NA,
        "peak_pos_err_days": pd.NA,
        "mass_early_ml1": pd.NA,
        "mass_tail_ml1": pd.NA,
        "mass_early_real": pd.NA,
        "mass_tail_real": pd.NA,
        "mass_early_diff": pd.NA,
        "mass_tail_diff": pd.NA,
    }

    if "rel_pos" in sub.columns:
        rel = pd.to_numeric(sub["rel_pos"], errors="coerce").to_numpy(float)
        out["mass_early_ml1"] = float(ml1[rel <= 0.15].sum())
        out["mass_tail_ml1"] = float(ml1[rel >= 0.85].sum())

    if "share_real" not in sub.columns or sub["share_real"].notna().sum() == 0:
        return out

    real = pd.to_numeric(sub["share_real"], errors="coerce").fillna(0.0).to_numpy(float)
    real = np.clip(real, 0.0, None)
    real = real / (real.sum() + 1e-12)

    out["has_real"] = 1
    peak_real = int(np.argmax(real)) if len(real) else pd.NA
    out["peak_pos_err_days"] = (out["peak_idx_ml1"] - peak_real) if pd.notna(peak_real) else pd.NA
    out["l1_share"] = float(np.abs(real - ml1).sum())
    out["ks_cdf"] = float(np.max(np.abs(np.cumsum(real) - np.cumsum(ml1))))

    if "rel_pos" in sub.columns:
        rel = pd.to_numeric(sub["rel_pos"], errors="coerce").to_numpy(float)
        m_early_real = float(real[rel <= 0.15].sum())
        m_tail_real = float(real[rel >= 0.85].sum())
        out["mass_early_real"] = m_early_real
        out["mass_tail_real"] = m_tail_real
        out["mass_early_diff"] = float(out["mass_early_ml1"] - m_early_real) if pd.notna(out["mass_early_ml1"]
) else pd.NA
        out["mass_tail_diff"] = float(out["mass_tail_ml1"] - m_tail_real) if pd.notna(out["mass_tail_ml1"]) el
se pd.NA

    return out


# =========================
# Main
# =========================
def main() -> None:
    AUDIT_DIR.mkdir(parents=True, exist_ok=True)
    created_at = pd.Timestamp.utcnow()
    var_map = _load_var_map()

    # ---------- Load
    uni = read_parquet(UNIVERSE_PATH).copy()
    curva = read_parquet(CURVA_PATH).copy()
    dist = read_parquet(DIST_PATH).copy()
    oferta = read_parquet(OFERTA_PATH).copy()
    full = read_parquet(PRED_FULL_PATH).copy()
    prog = read_parquet(PROG_PATH).copy() if PROG_PATH.exists() else pd.DataFrame()
    # ---------- Canon UNIVERSE
    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_ml1")
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])
    if "stage" in uni.columns:
        uni["stage"] = _canon_str(uni["stage"])
        uni = uni[uni["stage"].eq("HARVEST")].copy()
    uni = _ensure_relpos(uni)

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key].drop_duplicates()

    # ---------- Canon CURVA
    _require(curva, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "pred_factor_curva_ml1")
    curva["ciclo_id"] = curva["ciclo_id"].astype(str)
    curva["fecha"] = _to_date(curva["fecha"])
    curva["bloque_base"] = _canon_int(curva["bloque_base"])
    curva["variedad_canon"] = _canon_str(curva["variedad_canon"])
    curva_k = curva[key].drop_duplicates()

    # ---------- Canon DIST (día-level coverage; grade not needed here)
    _require(dist, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_ml1"], "pred_d
ist_grado_ml1")
    dist["ciclo_id"] = dist["ciclo_id"].astype(str)
    dist["fecha"] = _to_date(dist["fecha"])
    dist["bloque_base"] = _canon_int(dist["bloque_base"])
    dist["variedad_canon"] = _canon_str(dist["variedad_canon"])
    dist["grado"] = _canon_int(dist["grado"])
    dist_day_k = dist[key].drop_duplicates()

    # ---------- Canon OFERTA (baseline)
    _require(oferta, ["ciclo_id", "fecha", "bloque_base", "tallos_pred"], "pred_oferta_dia")
    oferta["ciclo_id"] = oferta["ciclo_id"].astype(str)
    oferta["fecha"] = _to_date(oferta["fecha"])
    oferta["bloque_base"] = _canon_int(oferta["bloque_base"])
    if "stage" in oferta.columns:
        oferta["stage"] = _canon_str(oferta["stage"])
        oferta = oferta[oferta["stage"].eq("HARVEST")].copy()
    # variedad_canon
    if "variedad_canon" in oferta.columns:
        oferta["variedad_canon"] = _canon_str(oferta["variedad_canon"])
    elif "variedad" in oferta.columns:
        oferta["variedad_canon"] = _canon_var_from_col(oferta, "variedad", var_map)
    else:
        oferta["variedad_canon"] = "UNKNOWN"

    oferta["tallos_pred_baseline_dia"] = pd.to_numeric(oferta["tallos_pred"], errors="coerce").fillna(0.0)
    oferta_k = oferta[key + ["tallos_pred_baseline_dia"]].drop_duplicates(subset=key)

    # ---------- Canon PROG (real)
    has_prog = (len(prog) > 0)
    if has_prog:
        _require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "dim_cosecha_progr
ess_bloque_fecha")
        prog["ciclo_id"] = prog["ciclo_id"].astype(str)
        prog["fecha"] = _to_date(prog["fecha"])
        prog["bloque_base"] = _canon_int(prog["bloque_base"])
        prog["variedad_canon"] = _canon_var_from_col(prog, "variedad", var_map)
        prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce")
        prog_k = prog[key + ["tallos_real_dia", "pct_avance_real"]].drop_duplicates(subset=key)
    else:
        prog_k = pd.DataFrame(columns=key + ["tallos_real_dia", "pct_avance_real"])

    # =========================
    # 1) COVERAGE
    # =========================
    miss_curva = uni_k.merge(curva_k, on=key, how="left", indicator=True)
    miss_curva = miss_curva[miss_curva["_merge"].eq("left_only")].drop(columns=["_merge"])
    miss_dist = uni_k.merge(dist_day_k, on=key, how="left", indicator=True)
    miss_dist = miss_dist[miss_dist["_merge"].eq("left_only")].drop(columns=["_merge"])

    miss_oferta = uni_k.merge(oferta_k[key].drop_duplicates(), on=key, how="left", indicator=True)
    miss_oferta = miss_oferta[miss_oferta["_merge"].eq("left_only")].drop(columns=["_merge"])

    miss_prog = uni_k.merge(prog_k[key].drop_duplicates(), on=key, how="left", indicator=True)
    miss_prog = miss_prog[miss_prog["_merge"].eq("left_only")].drop(columns=["_merge"])

    cov = pd.DataFrame([{
        "created_at": created_at,
        "universe_rows": int(len(uni_k)),
        "curva_rows": int(len(curva_k)),
        "dist_day_rows": int(len(dist_day_k)),
        "oferta_rows": int(len(oferta_k)),
        "prog_rows": int(len(prog_k)),
        "miss_curva_rows": int(len(miss_curva)),
        "miss_dist_rows": int(len(miss_dist)),
        "miss_oferta_rows": int(len(miss_oferta)),
        "miss_prog_rows": int(len(miss_prog)),
        "miss_curva_rate": float(len(miss_curva) / max(len(uni_k), 1)),
        "miss_dist_rate": float(len(miss_dist) / max(len(uni_k), 1)),
        "miss_oferta_rate": float(len(miss_oferta) / max(len(uni_k), 1)),
        "miss_prog_rate": float(len(miss_prog) / max(len(uni_k), 1)),
    }])
    write_parquet(cov, AUDIT_DIR / "coverage.parquet")

    # =========================
    # 2) INVARIANTS on FINAL
    # =========================
    _require(full, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"], "pred_tallos_grado_dia_ml1
_full")
    for c in ["tallos_pred_baseline_dia", "tallos_pred_ml1_dia",
              "tallos_pred_baseline_grado_dia", "tallos_pred_ml1_grado_dia"]:
        if c in full.columns:
            full[c] = pd.to_numeric(full[c], errors="coerce")

    full["ciclo_id"] = full["ciclo_id"].astype(str)
    full["fecha"] = _to_date(full["fecha"])
    full["bloque_base"] = _canon_int(full["bloque_base"])
    full["variedad_canon"] = _canon_str(full["variedad_canon"])
    full["grado"] = _canon_int(full["grado"])

    g_day = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    sum_base = full.groupby(g_day, dropna=False)["tallos_pred_baseline_grado_dia"].sum().rename("sum_grado_bas
e")
    sum_ml1 = full.groupby(g_day, dropna=False)["tallos_pred_ml1_grado_dia"].sum().rename("sum_grado_ml1")

    day_first = full.drop_duplicates(subset=g_day)[g_day + ["tallos_pred_baseline_dia", "tallos_pred_ml1_dia"]
].set_index(g_day)
    day_chk = day_first.join(sum_base).join(sum_ml1).reset_index()

    eps = 1e-6
    day_chk["mismatch_base"] = (day_chk["sum_grado_base"] - day_chk["tallos_pred_baseline_dia"]).abs() > eps
    day_chk["mismatch_ml1"] = (day_chk["sum_grado_ml1"] - day_chk["tallos_pred_ml1_dia"]).abs() > eps

    inv = pd.DataFrame([{
        "created_at": created_at,
        "pct_day_mismatch_base": float(day_chk["mismatch_base"].mean()) if len(day_chk) else np.nan,
        "pct_day_mismatch_ml1": float(day_chk["mismatch_ml1"].mean()) if len(day_chk) else np.nan,
        "pct_day_ml1_dia_nan": float(day_chk["tallos_pred_ml1_dia"].isna().mean()) if len(day_chk) else np.nan
,
        "pct_day_ml1_dia_zero": float((day_chk["tallos_pred_ml1_dia"].fillna(0.0) == 0.0).mean()) if len(day_c
hk) else np.nan,
    }])
    write_parquet(inv, AUDIT_DIR / "invariants_day.parquet")

    # Mass-balance per cycle (needs tallos_proy; take from oferta if present)
    if "tallos_proy" in oferta.columns:
        oferta_tp = oferta[["ciclo_id", "tallos_proy"]].copy()
        oferta_tp["ciclo_id"] = oferta_tp["ciclo_id"].astype(str)
        oferta_tp["tallos_proy"] = pd.to_numeric(oferta_tp["tallos_proy"], errors="coerce")
        tproy = oferta_tp.groupby("ciclo_id", dropna=False)["tallos_proy"].max()
    else:
        tproy = pd.Series(dtype=float)

    cyc = day_chk.groupby("ciclo_id", dropna=False).agg(
        ml1_sum=("tallos_pred_ml1_dia", "sum"),
        base_sum=("tallos_pred_baseline_dia", "sum"),
        days=("ciclo_id", "size"),
    )
    if len(tproy):
        cyc = cyc.join(tproy.rename("tallos_proy"), how="left")
        cyc["abs_diff_ml1_vs_proy"] = (cyc["ml1_sum"] - cyc["tallos_proy"]).abs()
        cyc["rel_diff_ml1_vs_proy"] = np.where(
            cyc["tallos_proy"].fillna(0).astype(float) > 0,
            cyc["abs_diff_ml1_vs_proy"] / cyc["tallos_proy"].astype(float),
            np.nan,
        )
    else:
        cyc["tallos_proy"] = np.nan
        cyc["abs_diff_ml1_vs_proy"] = np.nan
        cyc["rel_diff_ml1_vs_proy"] = np.nan

    cyc = cyc.reset_index()
    write_parquet(cyc, AUDIT_DIR / "mass_balance_cycle.parquet")

    # =========================
    # 3) SHAPE vs REAL (curve)
    # =========================
    # We approximate ML1 share from final output:
    # share_ml1_day = tallos_pred_ml1_dia / sum_cycle(tallos_pred_ml1_dia)
    # share_real_day = tallos_real_dia / sum_cycle(tallos_real_dia)
    day_panel = day_chk.merge(
        uni.drop_duplicates(subset=g_day)[g_day + [c for c in ["rel_pos", "day_in_harvest", "n_harvest_days",
"month"] if c in uni.columns]],
        on=g_day,
        how="left",
    ).merge(
        prog_k[g_day + ["tallos_real_dia", "pct_avance_real"]],
        on=g_day,
        how="left",
    )

    denom_ml1 = day_panel.groupby("ciclo_id", dropna=False)["tallos_pred_ml1_dia"].transform("sum")
    day_panel["share_ml1"] = np.where(denom_ml1 > 0, day_panel["tallos_pred_ml1_dia"] / denom_ml1, 0.0)

    denom_real = day_panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform(
        lambda s: np.nansum(pd.to_numeric(s, errors="coerce").to_numpy(dtype=float))
    )
    day_panel["share_real"] = np.where(denom_real > 0, day_panel["tallos_real_dia"] / denom_real, np.nan)

    write_parquet(day_panel, AUDIT_DIR / "day_panel.parquet")

    cyc_metrics = []
    for cid, sub in day_panel.groupby("ciclo_id", dropna=False):
        m = _cycle_shape_metrics(sub)
        m["ciclo_id"] = cid
        # tags for segmentation
        v = sub["variedad_canon"].dropna()
        m["variedad_canon"] = v.iloc[0] if len(v) else pd.NA
        b = sub["bloque_base"].dropna()
        m["bloque_base"] = b.iloc[0] if len(b) else pd.NA
        mo = sub["month"].dropna() if "month" in sub.columns else pd.Series([], dtype="Int64")
        m["month"] = int(mo.iloc[0]) if len(mo) else pd.NA
        cyc_metrics.append(m)

    cyc_shape = pd.DataFrame(cyc_metrics)
    write_parquet(cyc_shape, AUDIT_DIR / "shape_cycle.parquet")

    # segmentation summary (only where has_real)
    with_real = cyc_shape[cyc_shape["has_real"].eq(1)].copy()
    seg_rows = []
    if len(with_real):
        for by, tag in [
            (["variedad_canon"], "by_variedad"),
            (["bloque_base"], "by_bloque"),
            (["variedad_canon", "month"], "by_variedad_month"),
        ]:
            g = with_real.groupby(by, dropna=False).agg(
                n_cycles=("ciclo_id", "count"),
                l1_median=("l1_share", "median"),
                ks_median=("ks_cdf", "median"),
                peak_err_median=("peak_pos_err_days", "median"),
                early_diff_median=("mass_early_diff", "median"),
                tail_diff_median=("mass_tail_diff", "median"),
            ).reset_index()
            g["segment_tag"] = tag
            seg_rows.append(g.sort_values("n_cycles", ascending=False))
    seg = pd.concat(seg_rows, ignore_index=True) if seg_rows else pd.DataFrame()
    write_parquet(seg, AUDIT_DIR / "shape_segmentation.parquet")

    # =========================
    # Prints (state snapshot)
    # =========================
    print("\n=== COVERAGE ===")
    print(cov.to_string(index=False))

    print("\n=== INVARIANTS (day) ===")
    print(inv.to_string(index=False))

    if len(cyc):
        mx = float(pd.to_numeric(cyc["abs_diff_ml1_vs_proy"], errors="coerce").max())
        print("\n=== MASS BALANCE (cycle) ===")
        print(f"cycles={len(cyc):,} | max abs diff ml1 vs proy: {mx:.12f}")

    print("\n=== SHAPE (cycle) ===")
    print(f"cycles total={len(cyc_shape):,} | cycles with real={int(cyc_shape['has_real'].sum()):,}")
    if len(with_real):
        print("\nL1 share (lower is better):")
        print(with_real["l1_share"].describe().to_string())
        print("\nKS CDF (lower is better):")
        print(with_real["ks_cdf"].describe().to_string())
        print("\nPeak position error (days):")
        print(pd.to_numeric(with_real["peak_pos_err_days"], errors="coerce").describe().to_string())
        print("\nMass early diff (ML1 - real):")
        print(pd.to_numeric(with_real["mass_early_diff"], errors="coerce").describe().to_string())
        print("\nMass tail diff (ML1 - real):")
        print(pd.to_numeric(with_real["mass_tail_diff"], errors="coerce").describe().to_string())
    else:
        print("[WARN] No hay ciclos con real en el join (revisa PROG).")

    print(f"\nOK -> auditoría escrita en: {AUDIT_DIR}")


if __name__ == "__main__":
    main()

-----------------------------------------------
[9/65] FILE: \src\audit\audit_peso_tallo_ml2.py
-----------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
```

```
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"

IN_FACT = EVAL / "backtest_factor_ml2_peso_tallo_grado_dia.parquet"
IN_TALLOS = SILVER / "fact_cosecha_real_grado_dia.parquet"

TS = datetime.now().strftime("%Y%m%d_%H%M%S")

OUT_KPI_GLOBAL = EVAL / f"audit_peso_tallo_ml2_kpi_global_{TS}.parquet"
OUT_DIST = EVAL / f"audit_peso_tallo_ml2_ratio_dist_{TS}.parquet"
OUT_BY_GRADO = EVAL / f"audit_peso_tallo_ml2_kpi_by_grado_{TS}.parquet"
OUT_EXAMPLES = EVAL / f"audit_peso_tallo_ml2_examples_10x2_{TS}.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan


def wape_weighted(err_abs: pd.Series, w: pd.Series) -> float:
    err_abs = pd.to_numeric(err_abs, errors="coerce").fillna(0.0)
    w = pd.to_numeric(w, errors="coerce").fillna(0.0)
    denom = float(np.sum(w))
    if denom <= 0:
        return np.nan
    return float(np.sum(err_abs * w) / denom)


def main() -> None:
    df = read_parquet(IN_FACT).copy()

    # Tallos reales para ponderar (impacto en kg y wape ponderado)
    tl = read_parquet(IN_TALLOS).copy()
    tl["fecha"] = _to_date(tl["fecha"])
    tl["grado"] = _canon_str(tl["grado"]) if "grado" in tl.columns else tl["grado"]

    if "bloque_base" in tl.columns:
        tl["bloque_base"] = _canon_str(tl["bloque_base"])
    elif "bloque_padre" in tl.columns:
        tl["bloque_base"] = _canon_str(tl["bloque_padre"])
    elif "bloque" in tl.columns:
        tl["bloque_base"] = _canon_str(tl["bloque"])
    else:
        raise KeyError("fact_cosecha_real_grado_dia no tiene bloque_base/bloque_padre/bloque")

    tl["tallos_real"] = pd.to_numeric(tl["tallos_real"], errors="coerce").fillna(0.0)
    tl = tl.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"))

    # Canon fact
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_str(df["bloque_base"])
    df["grado"] = _canon_str(df["grado"])

    for c in ["peso_tallo_ml1_g", "peso_tallo_real_g", "peso_tallo_final_g", "pred_ratio_peso"]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce")

    # Merge tallos weights
    df = df.merge(tl, on=["fecha", "bloque_base", "grado"], how="left")
    df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce").fillna(0.0)

    # Keep only rows where we have real
    df = df[df["peso_tallo_real_g"].notna()].copy()
    # Errors (g/tallo)
    df["err_ml1_g"] = df["peso_tallo_real_g"] - df["peso_tallo_ml1_g"]
    df["err_ml2_g"] = df["peso_tallo_real_g"] - df["peso_tallo_final_g"]

    df["abs_err_ml1_g"] = df["err_ml1_g"].abs()
    df["abs_err_ml2_g"] = df["err_ml2_g"].abs()

    # Equivalent absolute kg error = abs_err_g * tallos / 1000
    df["abs_err_ml1_kg_equiv"] = (df["abs_err_ml1_g"] * df["tallos_real"]) / 1000.0
    df["abs_err_ml2_kg_equiv"] = (df["abs_err_ml2_g"] * df["tallos_real"]) / 1000.0

    # === KPI GLOBAL ===
    kpi_g = pd.DataFrame([{
        "n_rows": int(len(df)),
        "n_cycles": int(df["ciclo_id"].nunique()) if "ciclo_id" in df.columns else np.nan,
        "mae_ml1_g": mae(df["err_ml1_g"]),
        "mae_ml2_g": mae(df["err_ml2_g"]),
        "wape_wt_ml1_g": wape_weighted(df["abs_err_ml1_g"], df["tallos_real"]),
        "wape_wt_ml2_g": wape_weighted(df["abs_err_ml2_g"], df["tallos_real"]),
        "kg_abs_err_ml1": float(df["abs_err_ml1_kg_equiv"].sum()),
        "kg_abs_err_ml2": float(df["abs_err_ml2_kg_equiv"].sum()),
        "improvement_abs_mae_g": mae(df["err_ml1_g"]) - mae(df["err_ml2_g"]),
        "improvement_abs_wape_wt_g": wape_weighted(df["abs_err_ml1_g"], df["tallos_real"]) - wape_weighted(df[
"abs_err_ml2_g"], df["tallos_real"]),
        "improvement_abs_kg": float(df["abs_err_ml1_kg_equiv"].sum() - df["abs_err_ml2_kg_equiv"].sum()),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])
    write_parquet(kpi_g, OUT_KPI_GLOBAL)

    # === DISTRIBUCIÓN DE RATIOS ===
    r = pd.to_numeric(df["pred_ratio_peso"], errors="coerce")
    dist = pd.DataFrame([{
        "n": int(r.notna().sum()),
        "ratio_min": float(np.nanmin(r)),
        "ratio_p05": float(np.nanpercentile(r, 5)),
        "ratio_p25": float(np.nanpercentile(r, 25)),
        "ratio_median": float(np.nanmedian(r)),
        "ratio_p75": float(np.nanpercentile(r, 75)),
        "ratio_p95": float(np.nanpercentile(r, 95)),
        "ratio_max": float(np.nanmax(r)),
    }])
    write_parquet(dist, OUT_DIST)

    # === KPI POR GRADO (para ver sesgos por calidad) ===
    rows = []
    for grado, g in df.groupby("grado"):
        rows.append({
            "grado": str(grado),
            "n_rows": int(len(g)),
            "mae_ml1_g": mae(g["err_ml1_g"]),
            "mae_ml2_g": mae(g["err_ml2_g"]),
            "wape_wt_ml1_g": wape_weighted(g["abs_err_ml1_g"], g["tallos_real"]),
            "wape_wt_ml2_g": wape_weighted(g["abs_err_ml2_g"], g["tallos_real"]),
            "kg_abs_err_ml1": float(g["abs_err_ml1_kg_equiv"].sum()),
            "kg_abs_err_ml2": float(g["abs_err_ml2_kg_equiv"].sum()),
            "improvement_abs_kg": float(g["abs_err_ml1_kg_equiv"].sum() - g["abs_err_ml2_kg_equiv"].sum()),
        })
    by_grado = pd.DataFrame(rows).sort_values("improvement_abs_kg", ascending=False)
    write_parquet(by_grado, OUT_BY_GRADO)

    # === EJEMPLOS 10x2 (mejora vs empeora) por ciclo usando impacto kg ===
    if "ciclo_id" in df.columns:
        by_cycle = (
            df.groupby("ciclo_id", as_index=False)
              .agg(
                  kg_abs_err_ml1=("abs_err_ml1_kg_equiv", "sum"),
                  kg_abs_err_ml2=("abs_err_ml2_kg_equiv", "sum"),
                  n_rows=("fecha", "count"),
              )
        )
        by_cycle["improvement_kg"] = by_cycle["kg_abs_err_ml1"] - by_cycle["kg_abs_err_ml2"]
        top_good = by_cycle.sort_values("improvement_kg", ascending=False).head(10).assign(sample_group="TOP_I
MPROVE_10")
        top_bad = by_cycle.sort_values("improvement_kg", ascending=True).head(10).assign(sample_group="TOP_WOR
SE_10")
        ex = pd.concat([top_good, top_bad], ignore_index=True)

        # attach some context (bloque_base, variedad_canon) if available
        ctx_cols = [c for c in ["ciclo_id", "bloque_base", "variedad_canon", "area", "tipo_sp", "estado"] if c
 in df.columns]
        if ctx_cols:
            ctx = df[ctx_cols].drop_duplicates("ciclo_id")
            ex = ex.merge(ctx, on="ciclo_id", how="left")

        write_parquet(ex, OUT_EXAMPLES)
    else:
        ex = pd.DataFrame()
        write_parquet(ex, OUT_EXAMPLES)

    # === PRINT ===
    print("\n=== ML2 PESO POR TALLO AUDIT ===")
    print(f"KPI global parquet : {OUT_KPI_GLOBAL}")
    print(f"Ratio dist parquet : {OUT_DIST}")
    print(f"KPI por grado      : {OUT_BY_GRADO}")
    print(f"Examples 10x2      : {OUT_EXAMPLES}")
    print("\n--- KPI GLOBAL ---")
    print(kpi_g.to_string(index=False))
    print("\n--- RATIO DIST ---")
    print(dist.to_string(index=False))
    print("\n--- KPI POR GRADO (top 10 por mejora kg) ---")
    print(by_grado.head(10).to_string(index=False))
    if not ex.empty:
        print("\n--- EXAMPLES 10x2 ---")
        print(ex.to_string(index=False))


if __name__ == "__main__":
    main()

----------------------------------------------------------
[10/65] FILE: \src\audit\audit_poscosecha_ml2_baseline.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

IN_GLOBAL = EVAL / "ml2_poscosecha_baseline_eval_global.parquet"
IN_BY_DEST = EVAL / "ml2_poscosecha_baseline_eval_by_destino.parquet"
IN_BY_GR = EVAL / "ml2_poscosecha_baseline_eval_by_grado.parquet"
IN_COV = EVAL / "ml2_poscosecha_baseline_eval_coverage.parquet"

# Para ejemplos (top improve/worse) calculamos con el join directo
IN_PRED_FULL = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"
IN_REAL_HIDR_DH = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"
IN_REAL_MERMA_AJ = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"
```

```
OUT_KPI_GLOBAL = EVAL / "audit_poscosecha_baseline_kpi_global_{ts}.parquet"
OUT_KPI_DEST = EVAL / "audit_poscosecha_baseline_kpi_by_destino_{ts}.parquet"
OUT_KPI_GR = EVAL / "audit_poscosecha_baseline_kpi_by_grado_{ts}.parquet"
OUT_COV = EVAL / "audit_poscosecha_baseline_coverage_{ts}.parquet"
OUT_EXAMPLES = EVAL / "audit_poscosecha_baseline_examples_10x2_{ts}.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _as_of_date_default() -> pd.Timestamp:
    return pd.Timestamp.now().normalize() - pd.Timedelta(days=1)


def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    a = pd.to_numeric(a, errors="coerce")
    b = pd.to_numeric(b, errors="coerce")
    return a / b.replace(0, np.nan)


def main() -> None:
    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    aod = _as_of_date_default()

    g = read_parquet(IN_GLOBAL).copy()
    bd = read_parquet(IN_BY_DEST).copy()
    bg = read_parquet(IN_BY_GR).copy()
    cov = read_parquet(IN_COV).copy()

    # Persist snapshot
    write_parquet(g, Path(str(OUT_KPI_GLOBAL).format(ts=ts)))
    write_parquet(bd, Path(str(OUT_KPI_DEST).format(ts=ts)))
    write_parquet(bg, Path(str(OUT_KPI_GR).format(ts=ts)))
    write_parquet(cov, Path(str(OUT_COV).format(ts=ts)))

    # -------------------------
    # Examples (top improve/worse) sobre error de cajas_postcosecha_ml1 vs real "factor-based"
    # Nota: como no tenemos cajas_post real aquí, usamos un proxy robusto:
    #   score = |log_ratio_hidr| + |log_ratio_desp| + |log_ratio_ajuste| + (abs_err_dh_days/7)
    # y mostramos TOP 10 mejores (menor score) y TOP 10 peores (mayor score)
    # -------------------------
    pred = read_parquet(IN_PRED_FULL).copy()
    pred.columns = [str(c).strip() for c in pred.columns]
    pred["fecha"] = _to_date(pred["fecha"])
    pred["grado"] = _canon_int(pred["grado"])
    pred["destino"] = _canon_str(pred["destino"])

    # columnas pred
    dh_col = None
    for c in ["dh_dias_ml1", "dh_dias_pred_ml1", "dh_dias"]:
        if c in pred.columns:
            dh_col = c
            break
    fecha_post_pred_col = None
    for c in ["fecha_post_pred_ml1", "fecha_post_pred", "fecha_post_ml1"]:
        if c in pred.columns:
            fecha_post_pred_col = c
            break
    if fecha_post_pred_col:
        pred[fecha_post_pred_col] = _to_date(pred[fecha_post_pred_col])

    hidr_col = None
    for c in ["factor_hidr_ml1", "factor_hidr"]:
        if c in pred.columns:
            hidr_col = c
            break
    desp_col = None
    for c in ["factor_desp_ml1", "factor_desp"]:
        if c in pred.columns:
            desp_col = c
            break
    aj_col = None
    for c in ["ajuste_ml1", "factor_ajuste_ml1", "factor_ajuste"]:
        if c in pred.columns:
            aj_col = c
            break

    pred = pred[(pred["fecha"] <= aod)].copy()
    if fecha_post_pred_col:
        pred = pred[(pred[fecha_post_pred_col] <= aod)].copy()

    real_hd = read_parquet(IN_REAL_HIDR_DH).copy()
    real_hd.columns = [str(c).strip() for c in real_hd.columns]
    real_hd["fecha_cosecha"] = _to_date(real_hd["fecha_cosecha"])
    real_hd["fecha_post"] = _to_date(real_hd["fecha_post"])
    real_hd["grado"] = _canon_int(real_hd["grado"])
    real_hd["destino"] = _canon_str(real_hd["destino"])
    real_hd["tallos"] = pd.to_numeric(real_hd["tallos"], errors="coerce").fillna(0.0)
    real_hd["dh_dias"] = pd.to_numeric(real_hd["dh_dias"], errors="coerce")
    real_hd["peso_base_g"] = pd.to_numeric(real_hd["peso_base_g"], errors="coerce")
    real_hd["peso_post_g"] = pd.to_numeric(real_hd["peso_post_g"], errors="coerce")
    real_hd["factor_hidr_real"] = _safe_div(real_hd["peso_post_g"], real_hd["peso_base_g"])
    real_hd = real_hd[(real_hd["fecha_cosecha"] <= aod) & (real_hd["fecha_post"] <= aod)].copy()

    rg = (
        real_hd.groupby(["fecha_cosecha", "grado", "destino"], as_index=False)
        .agg(
            tallos=("tallos", "sum"),
            dh_real=("dh_dias", "median"),
            factor_hidr_real=("factor_hidr_real", "median"),
            fecha_post_real=("fecha_post", "median"),
        )
    )

    real_ma = read_parquet(IN_REAL_MERMA_AJ).copy()
    real_ma.columns = [str(c).strip() for c in real_ma.columns]
    real_ma["fecha_post"] = _to_date(real_ma["fecha_post"])
    real_ma["destino"] = _canon_str(real_ma["destino"])
    real_ma["factor_desp"] = pd.to_numeric(real_ma["factor_desp"], errors="coerce")
    real_ma["factor_ajuste"] = pd.to_numeric(real_ma["factor_ajuste"], errors="coerce")
    real_ma = real_ma[real_ma["fecha_post"] <= aod].copy()

    df = pred.merge(
        rg,
        left_on=["fecha", "grado", "destino"],
        right_on=["fecha_cosecha", "grado", "destino"],
        how="left",
    )

    df["fecha_post_key"] = df["fecha_post_real"]
    if fecha_post_pred_col:
        df["fecha_post_key"] = df["fecha_post_key"].where(df["fecha_post_key"].notna(), df[fecha_post_pred_col
])

    df = df.merge(
        real_ma.rename(columns={"fecha_post": "fecha_post_key"}),
        on=["fecha_post_key", "destino"],
        how="left",
    )

    # métricas por fila
    df["dh_pred"] = pd.to_numeric(df[dh_col], errors="coerce") if dh_col else np.nan
    df["abs_err_dh"] = (pd.to_numeric(df["dh_pred"], errors="coerce") - pd.to_numeric(df["dh_real"], errors="c
oerce")).abs()
```

```
    df["hidr_pred"] = pd.to_numeric(df[hidr_col], errors="coerce") if hidr_col else np.nan
    df["log_ratio_hidr"] = np.log(_safe_div(df["factor_hidr_real"], df["hidr_pred"]).clip(lower=1e-6))

    df["desp_pred"] = pd.to_numeric(df[desp_col], errors="coerce") if desp_col else np.nan
    df["log_ratio_desp"] = np.log(_safe_div(df["factor_desp"], df["desp_pred"]).clip(lower=1e-6))

    df["aj_pred"] = pd.to_numeric(df[aj_col], errors="coerce") if aj_col else np.nan
    df["log_ratio_aj"] = np.log(_safe_div(df["factor_ajuste"], df["aj_pred"]).clip(lower=1e-6))

    # score proxy (menor = mejor)
    df["score"] = (
        df["abs_err_dh"].fillna(0.0) / 7.0
        + df["log_ratio_hidr"].abs().fillna(0.0)
        + df["log_ratio_desp"].abs().fillna(0.0)
        + df["log_ratio_aj"].abs().fillna(0.0)
    )
    # ponderar por tallos para ranking
    df["wscore"] = df["score"] * pd.to_numeric(df["tallos"], errors="coerce").fillna(0.0).clip(lower=0.0)

    # agregamos por (fecha, grado, destino) para evitar que bloque domine por cantidad de filas
    key = ["fecha", "grado", "destino"]
    ex = (
        df.groupby(key, dropna=False, as_index=False)
          .agg(
              tallos=("tallos", "sum"),
              score=("score", "mean"),
              wscore=("wscore", "sum"),
              abs_err_dh=("abs_err_dh", "mean"),
              log_ratio_hidr=("log_ratio_hidr", "mean"),
              log_ratio_desp=("log_ratio_desp", "mean"),
              log_ratio_aj=("log_ratio_aj", "mean"),
              fecha_post_real=("fecha_post_real", "first"),
              fecha_post_key=("fecha_post_key", "first"),
          )
    )

    top_best = ex.sort_values(["wscore", "tallos"], ascending=[True, False]).head(10).copy()
    top_best["sample_group"] = "TOP_BEST_10"
    top_worst = ex.sort_values(["wscore", "tallos"], ascending=[False, False]).head(10).copy()
    top_worst["sample_group"] = "TOP_WORST_10"

    examples = pd.concat([top_best, top_worst], ignore_index=True)
    out_ex_path = Path(str(OUT_EXAMPLES).format(ts=ts))
    write_parquet(examples, out_ex_path)

    # -------------------------
    # Print
    # -------------------------
    print("\n=== POSCOSECHA ML2 BASELINE AUDIT ===")
    print(f"KPI global parquet : {Path(str(OUT_KPI_GLOBAL).format(ts=ts))}")
    print(f"KPI por destino    : {Path(str(OUT_KPI_DEST).format(ts=ts))}")
    print(f"KPI por grado      : {Path(str(OUT_KPI_GR).format(ts=ts))}")
    print(f"Coverage parquet   : {Path(str(OUT_COV).format(ts=ts))}")
    print(f"Examples 10x2      : {out_ex_path}")

    print("\n--- KPI GLOBAL ---")
    print(g.to_string(index=False))

    print("\n--- COVERAGE ---")
    print(cov.to_string(index=False))

    print("\n--- KPI DESTINO (head) ---")
    print(bd.head(10).to_string(index=False))

    print("\n--- KPI GRADO (head) ---")
    print(bg.head(10).to_string(index=False))

    print("\n--- EXAMPLES (10x2) ---")
    print(examples.to_string(index=False))


if __name__ == "__main__":
    main()

-------------------------------------------------
[11/65] FILE: \src\audit\audit_share_grado_ml2.py
-------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FACTOR = EVAL / "backtest_factor_ml2_share_grado.parquet"
IN_FINAL = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"
IN_REAL = SILVER / "fact_cosecha_real_grado_dia.parquet"
IN_PRED_ML1 = GOLD / "pred_tallos_grado_dia_ml1_full.parquet"

STAMP = datetime.now().strftime("%Y%m%d_%H%M%S")

OUT_KPI_GLOBAL = EVAL / f"audit_share_grado_ml2_kpi_global_{STAMP}.parquet"
OUT_KPI_BY_ESTADO = EVAL / f"audit_share_grado_ml2_kpi_by_estado_{STAMP}.parquet"
OUT_RATIO_DIST = EVAL / f"audit_share_grado_ml2_ratio_dist_{STAMP}.parquet"
OUT_SANITY = EVAL / f"audit_share_grado_ml2_sanity_{STAMP}.parquet"
OUT_EXAMPLES_10x2 = EVAL / f"audit_share_grado_ml2_examples_10x2_{STAMP}.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _resolve_block_base(df: pd.DataFrame) -> pd.DataFrame:
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
        return df
    if "bloque_padre" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_padre"])
        return df
    if "bloque" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque"])
        return df
    raise KeyError("No encuentro columna de bloque: bloque_base / bloque_padre / bloque")


def _safe_float(x) -> float:
    try:
        return float(x)
    except Exception:
        return float("nan")


def _wape(abs_err: pd.Series, w: pd.Series) -> float:
    a = pd.to_numeric(abs_err, errors="coerce").fillna(0.0).values
    ww = pd.to_numeric(w, errors="coerce").fillna(0.0).values
    denom = float(np.sum(ww))
    if denom <= 0:
        return float("nan")
    return float(np.sum(a * ww) / denom)


def main() -> None:
    fac = read_parquet(IN_FACTOR).copy()
    fin = read_parquet(IN_FINAL).copy()
    real = read_parquet(IN_REAL).copy()
    ml1 = read_parquet(IN_PRED_ML1).copy()

    # Canon
    for df in (fac, fin, ml1):
        df["fecha"] = _to_date(df["fecha"])
        df["bloque_base"] = _canon_str(df["bloque_base"])
        df["grado"] = _canon_str(df["grado"])
        if "variedad_canon" in df.columns:
            df["variedad_canon"] = _canon_str(df["variedad_canon"])

    real["fecha"] = _to_date(real["fecha"])
    real = _resolve_block_base(real)
    real["grado"] = _canon_str(real["grado"])
    real["tallos_real"] = pd.to_numeric(real["tallos_real"], errors="coerce").fillna(0.0)

    # ---- Share real por día/bloque ----
    rg = real.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"
))
    tot = rg.groupby(["fecha", "bloque_base"], as_index=False).agg(tallos_real_dia=("tallos_real", "sum"))
    rg = rg.merge(tot, on=["fecha", "bloque_base"], how="left")
    rg["share_real"] = np.where(rg["tallos_real_dia"] > 0, rg["tallos_real"] / rg["tallos_real_dia"], np.nan)

    # ---- ML1 (share + tallos pred por grado) ----
    ml1_need = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_ml1", "tallos_pred
_ml1_grado_dia"]
    ml1 = ml1[ml1_need].copy()
    ml1["share_grado_ml1"] = pd.to_numeric(ml1["share_grado_ml1"], errors="coerce")
    ml1["tallos_pred_ml1_grado_dia"] = pd.to_numeric(ml1["tallos_pred_ml1_grado_dia"], errors="coerce").fillna
(0.0)

    # ---- Factor: baseline share_ml1 y share_final ----
    fac_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]
    if "share_ml1" in fac.columns:
        fac_use = fac[fac_cols + ["share_ml1"]].copy()
    elif "share_grado_ml1" in fac.columns:
        fac_use = fac.rename(columns={"share_grado_ml1": "share_ml1"})[fac_cols + ["share_ml1"]].copy()
    else:
        raise KeyError(f"No encuentro share_ml1/share_grado_ml1 en factor. Columnas: {list(fac.columns)}")

    # fin YA trae share_final (del apply)
    need_fin = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_final", "tallos_final_gr
ado_dia"]
    for c in need_fin:
        if c not in fin.columns:
            raise KeyError(f"Falta columna en FINAL backtest: {c}. Columnas: {list(fin.columns)}")
    fin_use = fin[need_fin].copy()

    # ---- Dataset audit a grano ----
    df = fin_use.merge(fac_use, on=fac_cols, how="left")
    df = df.merge(ml1, on=fac_cols, how="left")
    df = df.merge(rg[["fecha", "bloque_base", "grado", "share_real", "tallos_real", "tallos_real_dia"]],
                  on=["fecha", "bloque_base", "grado"], how="left")

    # Bring estado if present in factor (optional)
    if "estado" in fac.columns:
        tmp = fac[fac_cols + ["estado"]].drop_duplicates(fac_cols)
        tmp["estado"] = _canon_str(tmp["estado"])
        df = df.merge(tmp, on=fac_cols, how="left")
    else:
        df["estado"] = "NA"

    # Cast
    df["share_ml1"] = pd.to_numeric(df["share_ml1"], errors="coerce")
    df["share_final"] = pd.to_numeric(df["share_final"], errors="coerce")
    df["share_real"] = pd.to_numeric(df["share_real"], errors="coerce")
    df["tallos_pred_ml1_grado_dia"] = pd.to_numeric(df["tallos_pred_ml1_grado_dia"], errors="coerce")
    df["tallos_final_grado_dia"] = pd.to_numeric(df["tallos_final_grado_dia"], errors="coerce")
    df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce")
    df["tallos_real_dia"] = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0)

    # Filter where share_real exists
    m = df["share_real"].notna() & df["share_ml1"].notna() & df["share_final"].notna()
    d = df.loc[m].copy()

    # Errors
    d["err_share_ml1"] = d["share_real"] - d["share_ml1"]
    d["err_share_ml2"] = d["share_real"] - d["share_final"]
    d["abs_err_share_ml1"] = d["err_share_ml1"].abs()
    d["abs_err_share_ml2"] = d["err_share_ml2"].abs()

    d["abs_err_tallos_ml1"] = (d["tallos_real"] - d["tallos_pred_ml1_grado_dia"]).abs()
    d["abs_err_tallos_ml2"] = (d["tallos_real"] - d["tallos_final_grado_dia"]).abs()

    # Ratios (diagnóstico)
    eps = 1e-6
    d["ratio_share_final_over_ml1"] = (d["share_final"] + eps) / (d["share_ml1"] + eps)

    # --- KPI GLOBAL ---
    w = d["tallos_real_dia"]
    kpi_global = pd.DataFrame([{
        "n_rows": int(len(d)),
        "n_cycles": int(d["ciclo_id"].nunique()),
        "mae_share_ml1": _safe_float(np.nanmean(d["abs_err_share_ml1"])),
        "mae_share_ml2": _safe_float(np.nanmean(d["abs_err_share_ml2"])),
        "wape_share_ml1_wt_day": _wape(d["abs_err_share_ml1"], w),
        "wape_share_ml2_wt_day": _wape(d["abs_err_share_ml2"], w),
        "mae_abs_tallos_ml1": _safe_float(np.nanmean(d["abs_err_tallos_ml1"])),
        "mae_abs_tallos_ml2": _safe_float(np.nanmean(d["abs_err_tallos_ml2"])),
        "improvement_abs_mae_share": _safe_float(np.nanmean(d["abs_err_share_ml1"]) - np.nanmean(d["abs_err_sh
are_ml2"])),
        "improvement_abs_mae_abs_tallos": _safe_float(np.nanmean(d["abs_err_tallos_ml1"]) - np.nanmean(d["abs_
err_tallos_ml2"])),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    # --- KPI POR ESTADO ---
    rows = []
    for estado, g in d.groupby("estado"):
        ww = g["tallos_real_dia"]
        rows.append({
            "estado": str(estado),
            "n_rows": int(len(g)),
            "n_cycles": int(g["ciclo_id"].nunique()),
            "mae_share_ml1": _safe_float(np.nanmean(g["abs_err_share_ml1"])),
            "mae_share_ml2": _safe_float(np.nanmean(g["abs_err_share_ml2"])),
            "wape_share_ml1_wt_day": _wape(g["abs_err_share_ml1"], ww),
            "wape_share_ml2_wt_day": _wape(g["abs_err_share_ml2"], ww),
            "mae_abs_tallos_ml1": _safe_float(np.nanmean(g["abs_err_tallos_ml1"])),
            "mae_abs_tallos_ml2": _safe_float(np.nanmean(g["abs_err_tallos_ml2"])),
            "improvement_abs_mae_share": _safe_float(np.nanmean(g["abs_err_share_ml1"]) - np.nanmean(g["abs_er
r_share_ml2"])),
        })
    kpi_by_estado = pd.DataFrame(rows).sort_values("improvement_abs_mae_share", ascending=False)

    # --- RATIO DIST ---
    r = d["ratio_share_final_over_ml1"].replace([np.inf, -np.inf], np.nan).dropna()
    ratio_dist = pd.DataFrame([{
        "n": int(len(r)),
        "ratio_min": _safe_float(r.min()),
        "ratio_p05": _safe_float(r.quantile(0.05)),
        "ratio_p25": _safe_float(r.quantile(0.25)),
        "ratio_median": _safe_float(r.median()),
        "ratio_p75": _safe_float(r.quantile(0.75)),
        "ratio_p95": _safe_float(r.quantile(0.95)),
        "ratio_max": _safe_float(r.max()),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    # --- SANITY: sum shares == 1 ---
    if "share_final" not in fin.columns:
        raise KeyError("FINAL no tiene share_final para sanity.")
    s = fin.groupby(["ciclo_id", "fecha"], as_index=False).agg(sum_share_final=("share_final", "sum"))
    sanity = pd.DataFrame([{
        "n_days": int(len(s)),
        "sum_share_min": _safe_float(s["sum_share_final"].min()),
        "sum_share_p25": _safe_float(s["sum_share_final"].quantile(0.25)),
        "sum_share_median": _safe_float(s["sum_share_final"].median()),
        "sum_share_p75": _safe_float(s["sum_share_final"].quantile(0.75)),
        "sum_share_max": _safe_float(s["sum_share_final"].max()),
        "pct_close_1pm_1e-6": _safe_float(np.mean(np.isclose(s["sum_share_final"].values, 1.0, atol=1e-6))),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    # --- EXAMPLES 10x2 (Top improve / Top worse) por reducción de error tallos ---
    # agregamos por ciclo_id el impacto total
    agg = d.groupby(["ciclo_id", "estado"], as_index=False).agg(
        abs_err_tallos_ml1=("abs_err_tallos_ml1", "sum"),
        abs_err_tallos_ml2=("abs_err_tallos_ml2", "sum"),
        n_rows=("ciclo_id", "size"),
    )
    agg["improvement_abs_tallos"] = agg["abs_err_tallos_ml1"] - agg["abs_err_tallos_ml2"]

    top = agg.sort_values("improvement_abs_tallos", ascending=False).head(10).copy()
    top["sample_group"] = "TOP_IMPROVE_10"
    worst = agg.sort_values("improvement_abs_tallos", ascending=True).head(10).copy()
    worst["sample_group"] = "TOP_WORSE_10"

    examples = pd.concat([top, worst], ignore_index=True)

    # attach representative bloque_base/variedad
    rep = fin[["ciclo_id", "bloque_base", "variedad_canon"]].drop_duplicates("ciclo_id")
    examples = examples.merge(rep, on="ciclo_id", how="left")

    # Write
    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(kpi_global, OUT_KPI_GLOBAL)
    write_parquet(kpi_by_estado, OUT_KPI_BY_ESTADO)
    write_parquet(ratio_dist, OUT_RATIO_DIST)
    write_parquet(sanity, OUT_SANITY)
    write_parquet(examples, OUT_EXAMPLES_10x2)

    print("\n=== ML2 SHARE GRADO AUDIT ===")
    print(f"KPI global parquet : {OUT_KPI_GLOBAL}")
    print(f"KPI por estado     : {OUT_KPI_BY_ESTADO}")
    print(f"Ratio dist parquet : {OUT_RATIO_DIST}")
    print(f"Sanity parquet     : {OUT_SANITY}")
    print(f"Examples 10x2      : {OUT_EXAMPLES_10x2}")

    print("\n--- KPI GLOBAL ---")
    print(kpi_global.to_string(index=False))

    print("\n--- KPI POR ESTADO ---")
    print(kpi_by_estado.to_string(index=False))

    print("\n--- RATIO DIST ---")
    print(ratio_dist.to_string(index=False))

    print("\n--- SANITY ---")
    print(sanity.to_string(index=False))

    print("\n--- EXAMPLES 10x2 ---")
    print(examples.sort_values(["sample_group", "improvement_abs_tallos"], ascending=[True, False]).to_string(
index=False))


if __name__ == "__main__":
    main()
```

```
--------------------------------------------------
[12/65] FILE: \src\audit\audit_tallos_curve_ml2.py
--------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"

IN_FACT = EVAL / "backtest_factor_ml2_tallos_curve_dia.parquet"
IN_CICLO = SILVER / "fact_ciclo_maestro.parquet"

OUT_KPI_GLOBAL = EVAL / f"audit_tallos_curve_ml2_kpi_global_{datetime.now():%Y%m%d_%H%M%S}.parquet"
OUT_KPI_ESTADO = EVAL / f"audit_tallos_curve_ml2_kpi_by_estado_{datetime.now():%Y%m%d_%H%M%S}.parquet"
OUT_DIST = EVAL / f"audit_tallos_curve_ml2_adjust_dist_{datetime.now():%Y%m%d_%H%M%S}.parquet"
OUT_EXAMPLES = EVAL / f"audit_tallos_curve_ml2_examples_10x2_{datetime.now():%Y%m%d_%H%M%S}.parquet"


def mae(s: pd.Series) -> float:
    return float(np.nanmean(np.abs(s))) if len(s) else np.nan


def wape(y: pd.Series, yhat: pd.Series) -> float:
    denom = np.sum(np.abs(y))
    return float(np.sum(np.abs(y - yhat)) / denom) if denom > 0 else np.nan


def main() -> None:
    df = read_parquet(IN_FACT).copy()
    ciclo = read_parquet(IN_CICLO)[["ciclo_id", "estado"]].drop_duplicates("ciclo_id")

    ciclo["estado"] = ciclo["estado"].astype(str).str.upper().str.strip()
    df = df.merge(ciclo, on="ciclo_id", how="left")

    # Core numeric
    for c in ["tallos_real_dia", "tallos_pred_ml1_dia", "tallos_final_ml2_dia", "pred_ratio"]:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

    df = df[(df["tallos_real_dia"] > 0) | (df["tallos_pred_ml1_dia"] > 0)].copy()

    # Errors
    df["err_ml1"] = df["tallos_real_dia"] - df["tallos_pred_ml1_dia"]
    df["err_ml2"] = df["tallos_real_dia"] - df["tallos_final_ml2_dia"]

    # === KPI GLOBAL ===
    kpi_g = pd.DataFrame([{
        "n_rows": len(df),
        "n_cycles": df["ciclo_id"].nunique(),
        "mae_ml1": mae(df["err_ml1"]),
        "mae_ml2": mae(df["err_ml2"]),
        "wape_ml1": wape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"]),
        "wape_ml2": wape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
        "improvement_abs_mae": mae(df["err_ml1"]) - mae(df["err_ml2"]),
        "improvement_abs_wape": wape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"])
                                - wape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])
    write_parquet(kpi_g, OUT_KPI_GLOBAL)
    # === DISTRIBUCIÓN AJUSTES ===
    r = df["pred_ratio"]
    dist = pd.DataFrame([{
        "n": len(r),
        "min": r.min(),
        "p05": np.percentile(r, 5),
        "p25": np.percentile(r, 25),
        "median": np.median(r),
        "p75": np.percentile(r, 75),
        "p95": np.percentile(r, 95),
        "max": r.max(),
        "pct_clip": float(((r <= np.exp(-1.5)) | (r >= np.exp(1.5))).mean()),
    }])
    write_parquet(dist, OUT_DIST)

    # === KPI POR ESTADO ===
    rows = []
    for estado, g in df.groupby("estado"):
        rows.append({
            "estado": estado,
            "n_rows": len(g),
            "n_cycles": g["ciclo_id"].nunique(),
            "mae_ml1": mae(g["err_ml1"]),
            "mae_ml2": mae(g["err_ml2"]),
            "wape_ml1": wape(g["tallos_real_dia"], g["tallos_pred_ml1_dia"]),
            "wape_ml2": wape(g["tallos_real_dia"], g["tallos_final_ml2_dia"]),
            "improvement_abs_wape": wape(g["tallos_real_dia"], g["tallos_pred_ml1_dia"])
                                     - wape(g["tallos_real_dia"], g["tallos_final_ml2_dia"]),
        })
    write_parquet(pd.DataFrame(rows), OUT_KPI_ESTADO)

    # === EJEMPLOS 10x2 ===
    by_cycle = (
        df.groupby("ciclo_id", as_index=False)
          .agg(
              estado=("estado", "first"),
              wape_ml1=("tallos_pred_ml1_dia", lambda x: wape(df.loc[x.index, "tallos_real_dia"], x)),
              wape_ml2=("tallos_final_ml2_dia", lambda x: wape(df.loc[x.index, "tallos_real_dia"], x)),
          )
    )
    by_cycle["improvement"] = by_cycle["wape_ml1"] - by_cycle["wape_ml2"]

    top = pd.concat([
        by_cycle.sort_values("improvement", ascending=False).head(10),
        by_cycle.sort_values("improvement", ascending=True).head(10),
    ])
    write_parquet(top, OUT_EXAMPLES)

    # === PRINT ===
    print("\n=== AUDIT ML2 ? CURVA DIARIA TALL0S ===")
    print(f"KPI global      : {OUT_KPI_GLOBAL}")
    print(f"KPI por estado  : {OUT_KPI_ESTADO}")
    print(f"Dist ajustes    : {OUT_DIST}")
    print(f"Ejemplos 10x2   : {OUT_EXAMPLES}")
    print("\n--- KPI GLOBAL ---")
    print(kpi_g.to_string(index=False))
    print("\n--- DIST AJUSTES ---")
    print(dist.to_string(index=False))


if __name__ == "__main__":
    main()

--------------------------------------------------------------------
[13/65] FILE: \src\bronze\build_conteo_tallos_alturas_gyp2419_raw.py
--------------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import yaml
```

```python
import numpy as np

from openpyxl import load_workbook

from common.io import write_parquet


def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)


def _table_to_df(path: Path, table_name: str) -> pd.DataFrame:
    """
    Lee una Excel Table (ListObject) por nombre (ej: 'Tabla6') desde cualquier hoja.
    Devuelve un DataFrame con headers correctos.
    """
    wb = load_workbook(filename=str(path), read_only=False, data_only=True)

    for ws in wb.worksheets:
        # ws.tables es dict: {table_name: Table}
        if table_name in ws.tables:
            tab = ws.tables[table_name]
            ref = tab.ref  # rango tipo "A1:H120"

            cells = ws[ref]
            data = [[c.value for c in row] for row in cells]
            if not data or len(data) < 2:
                return pd.DataFrame()

            header = [str(x).strip() if x is not None else "" for x in data[0]]
            rows = data[1:]

            df = pd.DataFrame(rows, columns=header)

            # limpieza básica de columnas vacías tipo "None" o ""
            df.columns = [str(c).strip() for c in df.columns]
            df = df.loc[:, ~pd.Series(df.columns).astype(str).str.match(r"^(none)?$", case=False)]
            return df

    raise ValueError(
        f"No encontré la tabla '{table_name}' en el archivo {path.name}. "
        f"Abre el Excel y confirma el nombre exacto de la tabla."
    )


def main():
    cfg = load_settings()
    src = cfg.get("sources", {})

    p_in = Path(src.get("conteo_tallos_alturas_gyp2419_path", ""))
    if not str(p_in).strip():
        raise ValueError("Config: falta sources.conteo_tallos_alturas_gyp2419_path")
    if not p_in.exists():
        raise FileNotFoundError(f"No existe archivo: {p_in}")

    table_name = src.get("conteo_tallos_alturas_gyp2419_table", "")
    if not table_name:
        raise ValueError("Config: falta sources.conteo_tallos_alturas_gyp2419_table (ej: 'Tabla6')")

    # output
    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)
    out_path = bronze_dir / "conteo_tallos_alturas_gyp2419_raw.parquet"

    df = _table_to_df(p_in, table_name)

    # metadata
    df["__source_file"] = str(p_in)
    df["__source_table"] = table_name
    df["__ingested_at"] = datetime.now().isoformat(timespec="seconds")
    # limpieza mínima de strings
    for c in df.columns:
        if df[c].dtype == object:
            df[c] = df[c].astype(str).str.strip().replace({"None": np.nan, "nan": np.nan})

    write_parquet(df, out_path)
    print(f"OK: {out_path} filas={len(df)} cols={len(df.columns)}")


if __name__ == "__main__":
    main()

------------------------------------------------
[14/65] FILE: \src\bronze\build_fenograma_raw.py
------------------------------------------------


-----------------------------------------------------------
[15/65] FILE: \src\eval\build_kpis_ajuste_poscosecha_ml2.py
-----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FINAL = GOLD / "pred_poscosecha_ml2_ajuste_grado_dia_bloque_destino_final.parquet"
IN_REAL_MA = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

OUT_G = EVAL / "ml2_ajuste_poscosecha_eval_global.parquet"
OUT_D = EVAL / "ml2_ajuste_poscosecha_eval_by_destino.parquet"
OUT_DIST = EVAL / "ml2_ajuste_poscosecha_eval_ratio_dist.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:
        if c in df.columns:
            return c
    raise KeyError("No encuentro fecha_post_pred_* en final.")


def _resolve_ajuste_ml1(df: pd.DataFrame) -> str:
    for c in ["factor_ajuste_ml1", "ajuste_ml1", "factor_ajuste_seed", "factor_ajuste"]:
        if c in df.columns:
            return c
    raise KeyError("No encuentro ajuste ML1 en final.")


def _weight_series(df: pd.DataFrame) -> pd.Series:
    for c in ["tallos_w", "tallos", "tallos_total_ml2", "tallos_total"]:
        if c in df.columns:
            return pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    return pd.Series(1.0, index=df.index, dtype="float64")


def _wmae_log_ratio(ratio: pd.Series, w: pd.Series) -> float:
    r = pd.to_numeric(ratio, errors="coerce")
    w = pd.to_numeric(w, errors="coerce").fillna(0.0)
    lr = np.log(r.replace(0, np.nan))
    m = lr.notna() & np.isfinite(lr) & w.notna() & (w >= 0)
    if not bool(m.any()):
        return np.nan
    denom = float(w[m].sum())
    if denom <= 0:
        return np.nan
    return float((lr[m].abs() * w[m]).sum() / denom)


def main() -> None:
    df = read_parquet(IN_FINAL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    df["fecha"] = _to_date(df["fecha"])
    df["destino"] = _canon_str(df["destino"])

    fecha_post_col = _resolve_fecha_post_pred(df)
    df[fecha_post_col] = _to_date(df[fecha_post_col])

    aj_ml1_col = _resolve_ajuste_ml1(df)
    df[aj_ml1_col] = pd.to_numeric(df[aj_ml1_col], errors="coerce")

    if "factor_ajuste_final" not in df.columns:
        raise KeyError("No encuentro factor_ajuste_final en pred_poscosecha_ml2_ajuste..._final.")

    df["factor_ajuste_final"] = pd.to_numeric(df["factor_ajuste_final"], errors="coerce")

    # real
    real = read_parquet(IN_REAL_MA).copy()
    real.columns = [str(c).strip() for c in real.columns]
    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])

    if "factor_ajuste" in real.columns:
        rc = "factor_ajuste"
    elif "ajuste" in real.columns:
        rc = "ajuste"
    else:
        raise ValueError("Real MA no trae factor_ajuste ni ajuste.")
    real[rc] = pd.to_numeric(real[rc], errors="coerce")

    real2 = (
        real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
            .agg(factor_ajuste_real=(rc, "median"))
    )

    df = df.merge(
        real2.rename(columns={"fecha_post": "fecha_post_key"}),
        left_on=[fecha_post_col, "destino"],
        right_on=["fecha_post_key", "destino"],
        how="left",
    )

    w = _weight_series(df)

    # solo con real
    m = df["factor_ajuste_real"].notna() & df[aj_ml1_col].notna() & df["factor_ajuste_final"].notna()
    d = df.loc[m].copy()
    if d.empty:
        raise ValueError("No hay filas con real para KPIs de ajuste.")

    w2 = _weight_series(d)
    # ratios (pred/real)
    d["ratio_ml1"] = d[aj_ml1_col] / d["factor_ajuste_real"].replace(0, np.nan)
    d["ratio_ml2"] = d["factor_ajuste_final"] / d["factor_ajuste_real"].replace(0, np.nan)

    out_g = pd.DataFrame([{
        "n_rows": int(len(d)),
        "n_dates": int(pd.to_datetime(d[fecha_post_col], errors="coerce").dt.normalize().nunique()),
        "mae_log_ml1_w": _wmae_log_ratio(d["ratio_ml1"], w2),
        "mae_log_ml2_w": _wmae_log_ratio(d["ratio_ml2"], w2),
        "median_ratio_ml1": float(np.nanmedian(pd.to_numeric(d["ratio_ml1"], errors="coerce").values)),
        "median_ratio_ml2": float(np.nanmedian(pd.to_numeric(d["ratio_ml2"], errors="coerce").values)),
        "p05_ratio_ml2": float(np.nanpercentile(pd.to_numeric(d["ratio_ml2"], errors="coerce").dropna().values
, 5)),
        "p95_ratio_ml2": float(np.nanpercentile(pd.to_numeric(d["ratio_ml2"], errors="coerce").dropna().values
, 95)),
        "improvement_abs_mae_log_w": float(_wmae_log_ratio(d["ratio_ml1"], w2) - _wmae_log_ratio(d["ratio_ml2"
], w2)),
        "created_at": pd.Timestamp(datetime.utcnow()).normalize(),
    }])

    rows = []
    for dest, g in d.groupby("destino"):
        ww = _weight_series(g)
        rows.append({
            "destino": str(dest),
            "n_rows": int(len(g)),
            "n_dates": int(pd.to_datetime(g[fecha_post_col], errors="coerce").dt.normalize().nunique()),
            "mae_log_ml1_w": _wmae_log_ratio(g["ratio_ml1"], ww),
            "mae_log_ml2_w": _wmae_log_ratio(g["ratio_ml2"], ww),
            "median_ratio_ml2": float(np.nanmedian(pd.to_numeric(g["ratio_ml2"], errors="coerce").values)),
            "improvement_abs_mae_log_w": float(_wmae_log_ratio(g["ratio_ml1"], ww) - _wmae_log_ratio(g["ratio_
ml2"], ww)),
        })
    out_d = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

    r = pd.to_numeric(d["ratio_ml2"], errors="coerce")
    out_dist = pd.DataFrame([{
        "n": int(len(d)),
        "ratio_min": float(np.nanmin(r.values)),
        "ratio_p05": float(np.nanpercentile(r.dropna().values, 5)),
        "ratio_p25": float(np.nanpercentile(r.dropna().values, 25)),
        "ratio_median": float(np.nanmedian(r.values)),
        "ratio_p75": float(np.nanpercentile(r.dropna().values, 75)),
        "ratio_p95": float(np.nanpercentile(r.dropna().values, 95)),
        "ratio_max": float(np.nanmax(r.values)),
        "created_at": pd.Timestamp(datetime.utcnow()).normalize(),
    }])

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(out_g, OUT_G)
    write_parquet(out_d, OUT_D)
    write_parquet(out_dist, OUT_DIST)

    print(f"[OK] Wrote global : {OUT_G}")
    print(out_g.to_string(index=False))
    print(f"\n[OK] Wrote destino: {OUT_D} rows={len(out_d)}")
    print(out_d.to_string(index=False))
    print(f"\n[OK] Wrote dist   : {OUT_DIST}")
    print(out_dist.to_string(index=False))


if __name__ == "__main__":
    main()

---------------------------------------------------------
[16/65] FILE: \src\eval\build_kpis_desp_poscosecha_ml2.py
---------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
```

```python
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"

IN_FINAL = DATA / "gold" / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

OUT_GLOBAL = EVAL / "ml2_desp_poscosecha_eval_global.parquet"
OUT_BY_DEST = EVAL / "ml2_desp_poscosecha_eval_by_destino.parquet"
OUT_DIST = EVAL / "ml2_desp_poscosecha_eval_ratio_dist.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def wmae_log_ratio(log_ratio: pd.Series, w: pd.Series) -> float:
    x = pd.to_numeric(log_ratio, errors="coerce")
    ww = pd.to_numeric(w, errors="coerce").fillna(0.0)
    m = x.notna() & (ww > 0)
    if not bool(m.any()):
        return float("nan")
    denom = float(ww[m].sum())
    if denom <= 0:
        return float(np.nanmean(np.abs(x[m].values)))
    return float((np.abs(x[m].values) * ww[m].values).sum() / denom)


def main() -> None:
    df = read_parquet(IN_FINAL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha_post_pred_used", "destino", "factor_desp_ml1", "factor_desp_final"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Final sin columnas: {sorted(miss)}")

    df["fecha_post_pred_used"] = _to_date(df["fecha_post_pred_used"])
    df["destino"] = _canon_str(df["destino"])
    df["factor_desp_ml1"] = pd.to_numeric(df["factor_desp_ml1"], errors="coerce")
    df["factor_desp_final"] = pd.to_numeric(df["factor_desp_final"], errors="coerce")

    # peso
    # peso (tallos). Si no existe tallos/tallos_w en el universo, usar peso=1.0 por fila
    if "tallos_w" in df.columns:
        w = pd.to_numeric(df["tallos_w"], errors="coerce")
    elif "tallos" in df.columns:
        w = pd.to_numeric(df["tallos"], errors="coerce")
    else:
        w = pd.Series(1.0, index=df.index)

    df["w"] = w.fillna(0.0)


    # Real
    real = read_parquet(IN_REAL).copy()
    real.columns = [str(c).strip() for c in real.columns]
    need_r = {"fecha_post", "destino", "factor_desp"}
    miss_r = need_r - set(real.columns)
    if miss_r:
        raise ValueError(f"Real sin columnas: {sorted(miss_r)}")

    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    real["factor_desp_real"] = pd.to_numeric(real["factor_desp"], errors="coerce")

    real2 = (
        real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
        .agg(factor_desp_real=("factor_desp_real", "median"))
    )

    x = df.merge(
        real2,
        left_on=["fecha_post_pred_used", "destino"],
        right_on=["fecha_post", "destino"],
        how="left",
    )

    eps = 1e-12
    x["ratio_ml1"] = (x["factor_desp_real"] + eps) / (x["factor_desp_ml1"] + eps)
    x["ratio_ml2"] = (x["factor_desp_real"] + eps) / (x["factor_desp_final"] + eps)
    x["log_ratio_ml1"] = np.log(x["ratio_ml1"])
    x["log_ratio_ml2"] = np.log(x["ratio_ml2"])

    m = x["factor_desp_real"].notna() & x["factor_desp_ml1"].notna() & x["factor_desp_final"].notna()
    d = x.loc[m].copy()

    out_g = pd.DataFrame(
        [
            {
                "n_rows": int(len(d)),
                "n_dates": int(d["fecha_post_pred_used"].nunique()),
                "mae_log_ml1_w": wmae_log_ratio(d["log_ratio_ml1"], d["w"]),
                "mae_log_ml2_w": wmae_log_ratio(d["log_ratio_ml2"], d["w"]),
                "median_ratio_ml1": float(np.nanmedian(d["ratio_ml1"].values)),
                "median_ratio_ml2": float(np.nanmedian(d["ratio_ml2"].values)),
                "p05_ratio_ml2": float(np.nanpercentile(d["ratio_ml2"].values, 5)),
                "p95_ratio_ml2": float(np.nanpercentile(d["ratio_ml2"].values, 95)),
                "improvement_abs_mae_log_w": wmae_log_ratio(d["log_ratio_ml1"], d["w"]) - wmae_log_ratio(d["lo
g_ratio_ml2"], d["w"]),
                "created_at": pd.Timestamp(datetime.now()).normalize(),
            }
        ]
    )

    rows = []
    for dest, g in d.groupby("destino"):
        rows.append(
            {
                "destino": str(dest),
                "n_rows": int(len(g)),
                "mae_log_ml1_w": wmae_log_ratio(g["log_ratio_ml1"], g["w"]),
                "mae_log_ml2_w": wmae_log_ratio(g["log_ratio_ml2"], g["w"]),
                "median_ratio_ml2": float(np.nanmedian(g["ratio_ml2"].values)),
                "improvement_abs_mae_log_w": wmae_log_ratio(g["log_ratio_ml1"], g["w"]) - wmae_log_ratio(g["lo
g_ratio_ml2"], g["w"]),
            }
        )
    out_d = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

    dist = pd.DataFrame(
        [
            {
                "n": int(len(d)),
                "ratio_min": float(np.nanmin(d["ratio_ml2"].values)),
                "ratio_p05": float(np.nanpercentile(d["ratio_ml2"].values, 5)),
                "ratio_p25": float(np.nanpercentile(d["ratio_ml2"].values, 25)),
                "ratio_median": float(np.nanmedian(d["ratio_ml2"].values)),
                "ratio_p75": float(np.nanpercentile(d["ratio_ml2"].values, 75)),
                "ratio_p95": float(np.nanpercentile(d["ratio_ml2"].values, 95)),
                "ratio_max": float(np.nanmax(d["ratio_ml2"].values)),
                "created_at": pd.Timestamp(datetime.now()).normalize(),
            }
        ]
    )

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(out_g, OUT_GLOBAL)
    write_parquet(out_d, OUT_BY_DEST)
    write_parquet(dist, OUT_DIST)

    print(f"[OK] Wrote global : {OUT_GLOBAL}")
    print(out_g.to_string(index=False))
    print(f"\n[OK] Wrote destino: {OUT_BY_DEST} rows={len(out_d)}")
    print(out_d.to_string(index=False))
    print(f"\n[OK] Wrote dist   : {OUT_DIST}")
    print(dist.to_string(index=False))


if __name__ == "__main__":
    main()

-------------------------------------------------------
[17/65] FILE: \src\eval\build_kpis_dh_poscosecha_ml2.py
-------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime

import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FINAL = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_GLOBAL = EVAL / "ml2_dh_poscosecha_eval_global.parquet"
OUT_BY_DESTINO = EVAL / "ml2_dh_poscosecha_eval_by_destino.parquet"
OUT_BY_GRADO = EVAL / "ml2_dh_poscosecha_eval_by_grado.parquet"
OUT_DIST = EVAL / "ml2_dh_poscosecha_eval_delta_dist.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _mae(x: pd.Series, w: pd.Series | None = None) -> float:
    x = pd.to_numeric(x, errors="coerce")
    if w is None:
        return float(np.nanmean(np.abs(x))) if len(x) else np.nan
    w = pd.to_numeric(w, errors="coerce").fillna(0.0).astype(float)
    m = x.notna() & np.isfinite(w)
    if not m.any():
        return np.nan
    ww = w[m].values
    ww = np.where(ww <= 0, 1.0, ww)
    return float(np.sum(np.abs(x[m].values) * ww) / np.sum(ww))


def _dist(s: pd.Series) -> dict:
    s = pd.to_numeric(s, errors="coerce").dropna()
    if s.empty:
        return {"n": 0}
    return {
        "n": int(len(s)),
        "min": float(s.min()),
        "p05": float(s.quantile(0.05)),
        "p25": float(s.quantile(0.25)),
        "median": float(s.median()),
        "p75": float(s.quantile(0.75)),
        "p95": float(s.quantile(0.95)),
        "max": float(s.max()),
    }


def main() -> None:
    fin = read_parquet(IN_FINAL).copy()
    real = read_parquet(IN_REAL).copy()
    fin.columns = [str(c).strip() for c in fin.columns]
    real.columns = [str(c).strip() for c in real.columns]

    fin["fecha"] = _to_date(fin["fecha"])
    fin["destino"] = _canon_str(fin["destino"])
    fin["grado"] = _canon_int(fin["grado"])

    # Real
    real["fecha_cosecha"] = _to_date(real["fecha_cosecha"])
    real["destino"] = _canon_str(real["destino"])
    real["grado"] = _canon_int(real["grado"])
    real["tallos"] = pd.to_numeric(real.get("tallos"), errors="coerce").fillna(0.0)
    real["dh_dias"] = _canon_int(real.get("dh_dias"))

    real_g = (
        real.groupby(["fecha_cosecha", "grado", "destino"], dropna=False, as_index=False)
            .agg(dh_real=("dh_dias", "median"), tallos_real=("tallos", "sum"))
    )

    df = fin.merge(
        real_g,
        left_on=["fecha", "grado", "destino"],
        right_on=["fecha_cosecha", "grado", "destino"],
        how="left",
    )

    # Errores
    df["dh_ml1"] = _canon_int(df.get("dh_ml1"))
    df["dh_dias_final"] = _canon_int(df.get("dh_dias_final"))
    df["dh_real"] = _canon_int(df.get("dh_real"))
    df["tallos_real"] = pd.to_numeric(df.get("tallos_real"), errors="coerce").fillna(0.0)

    m = df["dh_real"].notna() & df["dh_ml1"].notna() & df["dh_dias_final"].notna()
    d = df.loc[m].copy()

    d["err_ml1_days"] = (d["dh_real"].astype(float) - d["dh_ml1"].astype(float))
    d["err_ml2_days"] = (d["dh_real"].astype(float) - d["dh_dias_final"].astype(float))
    d["improvement_abs_days"] = d["err_ml1_days"].abs() - d["err_ml2_days"].abs()

    out_g = pd.DataFrame([{
        "n_rows": int(len(d)),
        "n_dates": int(d["fecha"].nunique()),
        "mae_ml1_days": _mae(d["err_ml1_days"], w=d["tallos_real"]),
        "mae_ml2_days": _mae(d["err_ml2_days"], w=d["tallos_real"]),
        "bias_ml1_days": float(np.nanmean(d["err_ml1_days"])),
        "bias_ml2_days": float(np.nanmean(d["err_ml2_days"])),
        "improvement_abs_mae_days": _mae(d["err_ml1_days"], w=d["tallos_real"]) - _mae(d["err_ml2_days"], w=d[
"tallos_real"]),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    rows_dest = []
    for dest, g in d.groupby("destino"):
        rows_dest.append({
            "destino": str(dest),
            "n_rows": int(len(g)),
            "mae_ml1_days": _mae(g["err_ml1_days"], w=g["tallos_real"]),
            "mae_ml2_days": _mae(g["err_ml2_days"], w=g["tallos_real"]),
            "bias_ml1_days": float(np.nanmean(g["err_ml1_days"])),
            "bias_ml2_days": float(np.nanmean(g["err_ml2_days"])),
            "improvement_abs_mae_days": _mae(g["err_ml1_days"], w=g["tallos_real"]) - _mae(g["err_ml2_days"],
w=g["tallos_real"]),
        })
    out_d = pd.DataFrame(rows_dest).sort_values("improvement_abs_mae_days", ascending=False)

    rows_gr = []
    for gr, g in d.groupby("grado"):
        rows_gr.append({
            "grado": int(gr) if pd.notna(gr) else None,
            "n_rows": int(len(g)),
            "mae_ml1_days": _mae(g["err_ml1_days"], w=g["tallos_real"]),
            "mae_ml2_days": _mae(g["err_ml2_days"], w=g["tallos_real"]),
            "bias_ml1_days": float(np.nanmean(g["err_ml1_days"])),
            "bias_ml2_days": float(np.nanmean(g["err_ml2_days"])),
            "improvement_abs_mae_days": _mae(g["err_ml1_days"], w=g["tallos_real"]) - _mae(g["err_ml2_days"],
w=g["tallos_real"]),
        })
    out_gd = pd.DataFrame(rows_gr).sort_values("improvement_abs_mae_days", ascending=False)

    dist = _dist(pd.to_numeric(fin.get("err_dh_days_pred_ml2"), errors="coerce"))
    out_dist = pd.DataFrame([{
        **dist,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(out_g, OUT_GLOBAL)
    write_parquet(out_d, OUT_BY_DESTINO)
    write_parquet(out_gd, OUT_BY_GRADO)
    write_parquet(out_dist, OUT_DIST)

    print(f"[OK] Wrote global : {OUT_GLOBAL}")
    print(out_g.to_string(index=False))
    print(f"\n[OK] Wrote destino: {OUT_BY_DESTINO} rows={len(out_d)}")
    print(out_d.to_string(index=False))
    print(f"\n[OK] Wrote grado  : {OUT_BY_GRADO} rows={len(out_gd)}")
    print(out_gd.head(10).to_string(index=False))
    print(f"\n[OK] Wrote dist   : {OUT_DIST}")
    print(out_dist.to_string(index=False))


if __name__ == "__main__":
    main()

--------------------------------------------------------
[18/65] FILE: \src\eval\build_kpis_harvest_horizon_ml2.py
--------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import argparse
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"

IN_FINAL_PROD = GOLD_DIR / "pred_harvest_horizon_final_ml2.parquet"
IN_FINAL_BT = EVAL_DIR / "backtest_pred_harvest_horizon_final_ml2.parquet"

OUT_PROD = EVAL_DIR / "ml2_harvest_horizon_eval_prod.parquet"
OUT_BT = EVAL_DIR / "ml2_harvest_horizon_eval_backtest.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else float("nan")


def _bias(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(x)) if len(x) else float("nan")


def _parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser()
    p.add_argument("--mode", choices=["prod", "backtest"], default="backtest")
    return p.parse_args()


def main() -> None:
    args = _parse_args()

    ciclo = read_parquet(IN_CICLO).copy()
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])
    ciclo["fecha_fin_cosecha"] = _to_date(ciclo["fecha_fin_cosecha"])

    if args.mode == "prod":
        pred = read_parquet(IN_FINAL_PROD).copy()
        out_path = OUT_PROD
    else:
        pred = read_parquet(IN_FINAL_BT).copy()
        out_path = OUT_BT

    # Canon dates
    pred["harvest_end_pred"] = _to_date(pred["harvest_end_pred"])
    pred["harvest_end_final"] = _to_date(pred["harvest_end_final"])

    pred["harvest_start_pred"] = _to_date(pred["harvest_start_pred"])
    pred["harvest_start_final"] = _to_date(pred["harvest_start_final"])

    # Merge
    df = ciclo.merge(pred, on="ciclo_id", how="inner")

    # Real duration
    df["n_harvest_days_real"] = (df["fecha_fin_cosecha"] - df["fecha_inicio_cosecha"]).dt.days + 1

    # Filter to valid rows
    df = df.loc[
        df["n_harvest_days_real"].notna()
        & df["n_harvest_days_pred"].notna()
        & df["n_harvest_days_final"].notna()
        & df["harvest_end_pred"].notna()
        & df["harvest_end_final"].notna(),
        :
    ].copy()

    df["n_harvest_days_real"] = pd.to_numeric(df["n_harvest_days_real"], errors="coerce")
    df["n_harvest_days_pred"] = pd.to_numeric(df["n_harvest_days_pred"], errors="coerce")
    df["n_harvest_days_final"] = pd.to_numeric(df["n_harvest_days_final"], errors="coerce")

    # Errors on duration
    df["err_n_days_ml1"] = df["n_harvest_days_real"] - df["n_harvest_days_pred"]
    df["err_n_days_ml2"] = df["n_harvest_days_real"] - df["n_harvest_days_final"]

    # Errors on end-date (in days)
    df["err_end_ml1_days"] = (df["fecha_fin_cosecha"] - df["harvest_end_pred"]).dt.days
    df["err_end_ml2_days"] = (df["fecha_fin_cosecha"] - df["harvest_end_final"]).dt.days

    # KPIs
    mae_ml1_n = _mae(df["err_n_days_ml1"])
    mae_ml2_n = _mae(df["err_n_days_ml2"])

    mae_ml1_end = _mae(df["err_end_ml1_days"])
    mae_ml2_end = _mae(df["err_end_ml2_days"])

    out = pd.DataFrame([{
        "mode": args.mode,
        "n": int(len(df)),
        "mae_n_days_ml1": mae_ml1_n,
        "mae_n_days_ml2": mae_ml2_n,
        "bias_n_days_ml1": _bias(df["err_n_days_ml1"]),
        "bias_n_days_ml2": _bias(df["err_n_days_ml2"]),
        "improvement_abs_n_days": (mae_ml1_n - mae_ml2_n) if (pd.notna(mae_ml1_n) and pd.notna(mae_ml2_n)) els
e np.nan,
        "mae_end_days_ml1": mae_ml1_end,
        "mae_end_days_ml2": mae_ml2_end,
        "bias_end_days_ml1": _bias(df["err_end_ml1_days"]),
        "bias_end_days_ml2": _bias(df["err_end_ml2_days"]),
        "improvement_abs_end_days": (mae_ml1_end - mae_ml2_end) if (pd.notna(mae_ml1_end) and pd.notna(mae_ml2
_end)) else np.nan,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    EVAL_DIR.mkdir(parents=True, exist_ok=True)
    write_parquet(out, out_path)

    print(f"[OK] Wrote eval: {out_path}")
    print(out.to_string(index=False))


if __name__ == "__main__":
    main()

-------------------------------------------------------
[19/65] FILE: \src\eval\build_kpis_harvest_start_ml2.py
-------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    # .../src/eval/file.py -> repo_root = parents[2]
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
```

```
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_FINAL = EVAL_DIR / "backtest_pred_harvest_start_final_ml2.parquet"

OUT_EVAL = EVAL_DIR / "ml2_harvest_start_eval.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def main() -> None:
    ciclo = read_parquet(IN_CICLO).copy()
    pred = read_parquet(IN_FINAL).copy()

    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])
    pred["harvest_start_pred"] = _to_date(pred["harvest_start_pred"])
    pred["harvest_start_final"] = _to_date(pred["harvest_start_final"])

    df = ciclo.merge(pred, on="ciclo_id", how="inner")
    df = df.loc[
        df["fecha_inicio_cosecha"].notna()
        & df["harvest_start_pred"].notna()
        & df["harvest_start_final"].notna(),
        :
    ].copy()

    df["err_ml1_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_pred"]).dt.days.astype(int)
    df["err_ml2_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_final"]).dt.days.astype(int)

    def _mae(x: pd.Series) -> float:
        return float(np.mean(np.abs(x))) if len(x) else float("nan")

    def _bias(x: pd.Series) -> float:
        return float(np.mean(x)) if len(x) else float("nan")

    mae_ml1 = _mae(df["err_ml1_days"])
    mae_ml2 = _mae(df["err_ml2_days"])

    out = pd.DataFrame([{
        "n": int(len(df)),
        "mae_ml1_days": mae_ml1,
        "mae_ml2_days": mae_ml2,
        "bias_ml1_days": _bias(df["err_ml1_days"]),
        "bias_ml2_days": _bias(df["err_ml2_days"]),
        "improvement_abs_days": (mae_ml1 - mae_ml2) if (pd.notna(mae_ml1) and pd.notna(mae_ml2)) else float("n
an"),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    EVAL_DIR.mkdir(parents=True, exist_ok=True)
    write_parquet(out, OUT_EVAL)
    print(f"[OK] Wrote eval: {OUT_EVAL}")


if __name__ == "__main__":
    main()

--------------------------------------------------------
[20/65] FILE: \src\eval\build_kpis_hidr_poscosecha_ml2.py
--------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FINAL = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_GLOBAL = EVAL / "ml2_hidr_poscosecha_eval_global.parquet"
OUT_BY_DESTINO = EVAL / "ml2_hidr_poscosecha_eval_by_destino.parquet"
OUT_BY_GRADO = EVAL / "ml2_hidr_poscosecha_eval_by_grado.parquet"
OUT_RATIO_DIST = EVAL / "ml2_hidr_poscosecha_eval_ratio_dist.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _factor_from_hidr_pct(hidr_pct: pd.Series) -> pd.Series:
    x = pd.to_numeric(hidr_pct, errors="coerce")
    return np.where(x.isna(), np.nan, np.where(x > 3.5, 1.0 + x / 100.0, x)).astype(float)


def _mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan


def main() -> None:
    df = read_parquet(IN_FINAL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"grado", "destino", "factor_hidr_ml1", "factor_hidr_final"}
    miss = need - set(df.columns)
    if miss:
        raise KeyError(f"Final sin columnas: {sorted(miss)}")

    # fecha_post_pred usada
    fpp = None
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml2", "fecha_post_pred_ml1", "
fecha_post_pred"]:
        if c in df.columns:
            fpp = c
            break
    if fpp is None:
        raise KeyError("No encuentro fecha_post_pred en final.")

    df[fpp] = _to_date(df[fpp])
    df["destino"] = _canon_str(df["destino"])
    df["grado"] = _canon_int(df["grado"])
    df["factor_hidr_ml1"] = pd.to_numeric(df["factor_hidr_ml1"], errors="coerce")
    df["factor_hidr_final"] = pd.to_numeric(df["factor_hidr_final"], errors="coerce")

    # real
    real = read_parquet(IN_REAL).copy()
    real.columns = [str(c).strip() for c in real.columns]
    if "hidr_pct" in real.columns:
        real["factor_hidr_real"] = _factor_from_hidr_pct(real["hidr_pct"])
    else:
        pb = pd.to_numeric(real.get("peso_base_g"), errors="coerce")
        pp = pd.to_numeric(real.get("peso_post_g"), errors="coerce")
        real["factor_hidr_real"] = np.where(pb > 0, pp / pb, np.nan)

    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    real["grado"] = _canon_int(real["grado"])

    # agregación real por llave
    if "tallos" in real.columns:
        real["tallos"] = pd.to_numeric(real["tallos"], errors="coerce").fillna(0.0)
        g = real.groupby(["fecha_post", "grado", "destino"], dropna=False)
        real2 = g.apply(
            lambda x: pd.Series({
                "factor_hidr_real": float(np.nansum(x["factor_hidr_real"] * x["tallos"]) / np.nansum(x["tallos
"])) if np.nansum(x["tallos"]) > 0 else float(np.nanmedian(x["factor_hidr_real"])),
                "tallos_real_sum": float(np.nansum(x["tallos"])),
            })
        ).reset_index()
    else:
        real2 = (
            real.groupby(["fecha_post", "grado", "destino"], dropna=False, as_index=False)
                .agg(factor_hidr_real=("factor_hidr_real", "median"))
        )
        real2["tallos_real_sum"] = np.nan

    mrg = df.merge(
        real2,
        left_on=[fpp, "grado", "destino"],
        right_on=["fecha_post", "grado", "destino"],
        how="left",
    ).drop(columns=["fecha_post"], errors="ignore")

    m = mrg["factor_hidr_real"].notna() & mrg["factor_hidr_ml1"].notna() & mrg["factor_hidr_final"].notna()
    d = mrg.loc[m].copy()
    if d.empty:
        raise ValueError("No hay match con real para evaluar hidratación.")

    eps = 1e-9
    d["ratio_ml1"] = d["factor_hidr_real"] / d["factor_hidr_ml1"].clip(lower=eps)
    d["ratio_ml2"] = d["factor_hidr_real"] / d["factor_hidr_final"].clip(lower=eps)

    d["log_ratio_ml1"] = np.log(d["ratio_ml1"].clip(lower=eps))
    d["log_ratio_ml2"] = np.log(d["ratio_ml2"].clip(lower=eps))

    # peso por tallos
    w = pd.to_numeric(d.get("tallos_real_sum"), errors="coerce").fillna(1.0).clip(lower=0.0)
    if (w.sum() <= 0) or (w.isna().all()):
        w = pd.Series(1.0, index=d.index)

    def wmae(x):
        x = pd.to_numeric(x, errors="coerce")
        mm = x.notna()
        if not mm.any():
            return np.nan
        return float(np.sum(np.abs(x[mm]) * w[mm]) / np.sum(w[mm]))

    out_g = pd.DataFrame([{
        "n_rows": int(len(d)),
        "n_dates": int(d[fpp].nunique()),
        "mae_log_ml1_w": wmae(d["log_ratio_ml1"]),
        "mae_log_ml2_w": wmae(d["log_ratio_ml2"]),
        "median_ratio_ml1": float(np.nanmedian(d["ratio_ml1"])),
        "median_ratio_ml2": float(np.nanmedian(d["ratio_ml2"])),
        "p05_ratio_ml2": float(np.nanpercentile(d["ratio_ml2"], 5)),
        "p95_ratio_ml2": float(np.nanpercentile(d["ratio_ml2"], 95)),
        "improvement_abs_mae_log_w": wmae(d["log_ratio_ml1"]) - wmae(d["log_ratio_ml2"]),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])
```

```python
# by destino
rows = []
for dest, g in d.groupby("destino"):
    ww = pd.to_numeric(g.get("tallos_real_sum"), errors="coerce").fillna(1.0).clip(lower=0.0)
    if ww.sum() <= 0:
        ww = pd.Series(1.0, index=g.index)

    def _wmae(v):
        v = pd.to_numeric(v, errors="coerce")
        mm = v.notna()
        return float(np.sum(np.abs(v[mm]) * ww[mm]) / np.sum(ww[mm])) if mm.any() else np.nan

    rows.append({
        "destino": str(dest),
        "n_rows": int(len(g)),
        "mae_log_ml1_w": _wmae(g["log_ratio_ml1"]),
        "mae_log_ml2_w": _wmae(g["log_ratio_ml2"]),
        "median_ratio_ml2": float(np.nanmedian(g["ratio_ml2"])),
        "improvement_abs_mae_log_w": _wmae(g["log_ratio_ml1"]) - _wmae(g["log_ratio_ml2"]),
    })
out_bd = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

# by grado
rows = []
for gr, g in d.groupby("grado"):
    ww = pd.to_numeric(g.get("tallos_real_sum"), errors="coerce").fillna(1.0).clip(lower=0.0)
    if ww.sum() <= 0:
        ww = pd.Series(1.0, index=g.index)

    def _wmae(v):
        v = pd.to_numeric(v, errors="coerce")
        mm = v.notna()
        return float(np.sum(np.abs(v[mm]) * ww[mm]) / np.sum(ww[mm])) if mm.any() else np.nan

    rows.append({
        "grado": int(gr) if pd.notna(gr) else None,
        "n_rows": int(len(g)),
        "mae_log_ml1_w": _wmae(g["log_ratio_ml1"]),
        "mae_log_ml2_w": _wmae(g["log_ratio_ml2"]),
        "median_ratio_ml2": float(np.nanmedian(g["ratio_ml2"])),
        "improvement_abs_mae_log_w": _wmae(g["log_ratio_ml1"]) - _wmae(g["log_ratio_ml2"]),
    })
out_bg = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

# ratio dist ml2
out_dist = pd.DataFrame([{
    "n": int(len(d)),
    "ratio_min": float(np.nanmin(d["ratio_ml2"])),
    "ratio_p05": float(np.nanpercentile(d["ratio_ml2"], 5)),
    "ratio_p25": float(np.nanpercentile(d["ratio_ml2"], 25)),
    "ratio_median": float(np.nanmedian(d["ratio_ml2"])),
    "ratio_p75": float(np.nanpercentile(d["ratio_ml2"], 75)),
    "ratio_p95": float(np.nanpercentile(d["ratio_ml2"], 95)),
    "ratio_max": float(np.nanmax(d["ratio_ml2"])),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
}])

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_GLOBAL)
write_parquet(out_bd, OUT_BY_DESTINO)
write_parquet(out_bg, OUT_BY_GRADO)
write_parquet(out_dist, OUT_RATIO_DIST)

print(f"[OK] Wrote global : {OUT_GLOBAL}")
print(out_g.to_string(index=False))
print(f"\n[OK] Wrote destino: {OUT_BY_DESTINO} rows={len(out_bd)}")
print(out_bd.to_string(index=False))
print(f"\n[OK] Wrote grado  : {OUT_BY_GRADO} rows={len(out_bg)}")
print(out_bg.head(13).to_string(index=False))
print(f"\n[OK] Wrote dist   : {OUT_RATIO_DIST}")
print(out_dist.to_string(index=False))
```

```
if __name__ == "__main__":
    main()

----------------------------------------------------
[21/65] FILE: \src\eval\build_kpis_peso_tallo_ml2.py
----------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"

IN_FACT = EVAL / "backtest_factor_ml2_peso_tallo_grado_dia.parquet"
IN_TALLOS = SILVER / "fact_cosecha_real_grado_dia.parquet"

OUT_GLOBAL = EVAL / "ml2_peso_tallo_eval_global.parquet"
OUT_ESTADO = EVAL / "ml2_peso_tallo_eval_by_estado.parquet"
OUT_RATIO = EVAL / "ml2_peso_tallo_eval_ratio_dist.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _mae(x: pd.Series) -> float:
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan


def _wape_weighted(err_abs: pd.Series, w: pd.Series) -> float:
    w = pd.to_numeric(w, errors="coerce").fillna(0.0)
    err_abs = pd.to_numeric(err_abs, errors="coerce").fillna(0.0)
    denom = float(np.sum(w))
    if denom <= 0:
        return np.nan
    return float(np.sum(err_abs * w) / denom)


def main() -> None:
    df = read_parquet(IN_FACT).copy()

    # Tallos reales (peso por tallo debe ponderarse por tallos)
    tl = read_parquet(IN_TALLOS).copy()
    tl["fecha"] = _to_date(tl["fecha"])
    tl["grado"] = _canon_str(tl["grado"]) if "grado" in tl.columns else tl["grado"]
    if "bloque_base" not in tl.columns:
        if "bloque_padre" in tl.columns:
            tl["bloque_base"] = _canon_str(tl["bloque_padre"])
        elif "bloque" in tl.columns:
            tl["bloque_base"] = _canon_str(tl["bloque"])
        else:
            raise KeyError("fact_cosecha_real_grado_dia no tiene bloque_base/bloque_padre/bloque")
    else:
        tl["bloque_base"] = _canon_str(tl["bloque_base"])
```

```
tl["tallos_real"] = pd.to_numeric(tl["tallos_real"], errors="coerce").fillna(0.0)
tl = tl.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"))

# Canon fact
df["fecha"] = _to_date(df["fecha"])
df["bloque_base"] = _canon_str(df["bloque_base"])
df["grado"] = _canon_str(df["grado"])
if "estado" in df.columns:
    df["estado"] = _canon_str(df["estado"])

for c in ["peso_tallo_ml1_g", "peso_tallo_real_g", "peso_tallo_final_g", "pred_ratio_peso"]:
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors="coerce")

df = df.merge(tl, on=["fecha", "bloque_base", "grado"], how="left")
df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce").fillna(0.0)

# Keep rows where we have real peso_tallo and some weight
df = df[df["peso_tallo_real_g"].notna()].copy()

# Errors in g/tallo
df["err_ml1_g"] = df["peso_tallo_real_g"] - df["peso_tallo_ml1_g"]
df["err_ml2_g"] = df["peso_tallo_real_g"] - df["peso_tallo_final_g"]

# Weighted absolute error (by tallos)
df["abs_err_ml1_g"] = df["err_ml1_g"].abs()
df["abs_err_ml2_g"] = df["err_ml2_g"].abs()

# Impact kg (approx)
df["abs_err_ml1_kg_equiv"] = (df["abs_err_ml1_g"] * df["tallos_real"]) / 1000.0
df["abs_err_ml2_kg_equiv"] = (df["abs_err_ml2_g"] * df["tallos_real"]) / 1000.0

out_g = pd.DataFrame([{
    "n_rows": int(len(df)),
    "n_cycles": int(df["ciclo_id"].nunique()) if "ciclo_id" in df.columns else np.nan,
    "mae_ml1_g": _mae(df["err_ml1_g"]),
    "mae_ml2_g": _mae(df["err_ml2_g"]),
    "wape_wt_ml1_g": _wape_weighted(df["abs_err_ml1_g"], df["tallos_real"]),
    "wape_wt_ml2_g": _wape_weighted(df["abs_err_ml2_g"], df["tallos_real"]),
    "kg_abs_err_ml1": float(df["abs_err_ml1_kg_equiv"].sum()),
    "kg_abs_err_ml2": float(df["abs_err_ml2_kg_equiv"].sum()),
    "improvement_abs_mae_g": _mae(df["err_ml1_g"]) - _mae(df["err_ml2_g"]),
    "improvement_abs_wape_wt_g": _wape_weighted(df["abs_err_ml1_g"], df["tallos_real"])
                                 - _wape_weighted(df["abs_err_ml2_g"], df["tallos_real"]),
    "improvement_abs_kg": float(df["abs_err_ml1_kg_equiv"].sum() - df["abs_err_ml2_kg_equiv"].sum()),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
}])

# Ratio dist sanity
r = pd.to_numeric(df["pred_ratio_peso"], errors="coerce")
out_r = pd.DataFrame([{
    "n": int(r.notna().sum()),
    "ratio_min": float(np.nanmin(r)),
    "ratio_p05": float(np.nanpercentile(r, 5)),
    "ratio_p25": float(np.nanpercentile(r, 25)),
    "ratio_median": float(np.nanmedian(r)),
    "ratio_p75": float(np.nanpercentile(r, 75)),
    "ratio_p95": float(np.nanpercentile(r, 95)),
    "ratio_max": float(np.nanmax(r)),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
}])

# By estado if exists
rows = []
if "estado" in df.columns and df["estado"].notna().any():
    for est, g in df.groupby("estado"):
        rows.append({
            "estado": str(est),
            "n_rows": int(len(g)),
            "mae_ml1_g": _mae(g["err_ml1_g"]),
            "mae_ml2_g": _mae(g["err_ml2_g"]),
            "wape_wt_ml1_g": _wape_weighted(g["abs_err_ml1_g"], g["tallos_real"]),
                "wape_wt_ml2_g": _wape_weighted(g["abs_err_ml2_g"], g["tallos_real"]),
                "kg_abs_err_ml1": float(g["abs_err_ml1_kg_equiv"].sum()),
                "kg_abs_err_ml2": float(g["abs_err_ml2_kg_equiv"].sum()),
                "improvement_abs_kg": float(g["abs_err_ml1_kg_equiv"].sum() - g["abs_err_ml2_kg_equiv"].sum())
,
            })
    out_e = pd.DataFrame(rows)

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(out_g, OUT_GLOBAL)
    write_parquet(out_r, OUT_RATIO)
    write_parquet(out_e, OUT_ESTADO)

    print(f"[OK] Wrote global: {OUT_GLOBAL}")
    print(out_g.to_string(index=False))
    print(f"\n[OK] Wrote ratio : {OUT_RATIO}")
    print(out_r.to_string(index=False))
    print(f"\n[OK] Wrote estado: {OUT_ESTADO} rows={len(out_e)}")
    if not out_e.empty:
        print(out_e.sort_values('improvement_abs_kg', ascending=False).to_string(index=False))


if __name__ == "__main__":
    main()

------------------------------------------------------------
[22/65] FILE: \src\eval\build_kpis_poscosecha_ml2_baseline.py
------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"
EVAL = DATA / "eval" / "ml2"

# =========================
# INPUTS
# =========================
IN_PRED_FULL = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"

# Real hidr + DH (por fecha_cosecha/fecha_post, grado, destino)
IN_REAL_HIDR_DH = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

# Real merma/ajuste (por fecha_post, destino)
IN_REAL_MERMA_AJ = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

# =========================
# OUTPUTS
# =========================
OUT_GLOBAL = EVAL / "ml2_poscosecha_baseline_eval_global.parquet"
OUT_BY_DESTINO = EVAL / "ml2_poscosecha_baseline_eval_by_destino.parquet"
OUT_BY_GRADO = EVAL / "ml2_poscosecha_baseline_eval_by_grado.parquet"
OUT_COVERAGE = EVAL / "ml2_poscosecha_baseline_eval_coverage.parquet"


# =========================
# HELPERS
# =========================
def _to_date(s: pd.Series) -> pd.Series:
```

```python
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _as_of_date_default() -> pd.Timestamp:
    # Regla operativa: "hoy" no se usa porque no está cerrado ? usar hoy-1
    return pd.Timestamp.now().normalize() - pd.Timedelta(days=1)


def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    a = pd.to_numeric(a, errors="coerce")
    b = pd.to_numeric(b, errors="coerce")
    return a / b.replace(0, np.nan)


def mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan


def wape(abs_err: pd.Series, y_true: pd.Series) -> float:
    abs_err = pd.to_numeric(abs_err, errors="coerce").fillna(0.0)
    y_true = pd.to_numeric(y_true, errors="coerce").fillna(0.0)
    denom = float(np.sum(np.abs(y_true)))
    if denom <= 0:
        return np.nan
    return float(np.sum(abs_err) / denom)


def _require_cols(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise KeyError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")


def main(as_of_date: str | None = None) -> None:
    AOD = _as_of_date_default() if as_of_date is None else pd.to_datetime(as_of_date).normalize()

    pred = read_parquet(IN_PRED_FULL).copy()
    pred.columns = [str(c).strip() for c in pred.columns]

    real_hd = read_parquet(IN_REAL_HIDR_DH).copy()
    real_hd.columns = [str(c).strip() for c in real_hd.columns]

    real_ma = read_parquet(IN_REAL_MERMA_AJ).copy()
    real_ma.columns = [str(c).strip() for c in real_ma.columns]

    # =========================
    # Canon pred
    # =========================
    _require_cols(pred, ["fecha", "grado", "destino"], "pred_poscosecha_ml2_full")

    pred["fecha"] = _to_date(pred["fecha"])
    pred["grado"] = _canon_int(pred["grado"])
    pred["destino"] = _canon_str(pred["destino"])

    # columnas ML1 esperadas (con fallback)
    # DH
    dh_col = None
    for c in ["dh_dias_ml1", "dh_dias_pred_ml1", "dh_dias"]:
        if c in pred.columns:
            dh_col = c
            break

    # fecha_post predicha (si existe, mejor)
    fecha_post_pred_col = None
    for c in ["fecha_post_pred_ml1", "fecha_post_pred", "fecha_post_ml1"]:
        if c in pred.columns:
            fecha_post_pred_col = c
            break

    if fecha_post_pred_col:
        pred[fecha_post_pred_col] = _to_date(pred[fecha_post_pred_col])
    elif dh_col:
        pred["fecha_post_pred_ml1"] = pred["fecha"] + pd.to_timedelta(
            pd.to_numeric(pred[dh_col], errors="coerce").fillna(0).astype(int), unit="D"
        )
        fecha_post_pred_col = "fecha_post_pred_ml1"
    else:
        raise KeyError("No encuentro dh ni fecha_post_pred en pred. Espero dh_dias_ml1 o fecha_post_pred_ml1."
)

    # Hidr
    hidr_col = None
    for c in ["factor_hidr_ml1", "factor_hidr"]:
        if c in pred.columns:
            hidr_col = c
            break

    # Desp
    desp_col = None
    for c in ["factor_desp_ml1", "factor_desp"]:
        if c in pred.columns:
            desp_col = c
            break

    # Ajuste
    aj_col = None
    for c in ["ajuste_ml1", "factor_ajuste_ml1", "factor_ajuste"]:
        if c in pred.columns:
            aj_col = c
            break

    # filtro as_of
    pred = pred[pred["fecha"] <= AOD].copy()
    pred = pred[pred[fecha_post_pred_col] <= AOD].copy()

    # =========================
    # Canon real hidr/dh
    # =========================
    _require_cols(
        real_hd,
        ["fecha_cosecha", "fecha_post", "dh_dias", "grado", "destino", "tallos", "peso_base_g", "peso_post_g"]
,
        "fact_hidratacion_real_post_grado_destino",
    )
    real_hd["fecha_cosecha"] = _to_date(real_hd["fecha_cosecha"])
    real_hd["fecha_post"] = _to_date(real_hd["fecha_post"])
    real_hd["grado"] = _canon_int(real_hd["grado"])
    real_hd["destino"] = _canon_str(real_hd["destino"])
    real_hd["tallos"] = pd.to_numeric(real_hd["tallos"], errors="coerce").fillna(0.0)
    real_hd["dh_dias"] = pd.to_numeric(real_hd["dh_dias"], errors="coerce")

    real_hd["peso_base_g"] = pd.to_numeric(real_hd["peso_base_g"], errors="coerce")
    real_hd["peso_post_g"] = pd.to_numeric(real_hd["peso_post_g"], errors="coerce")
    real_hd["factor_hidr_real"] = _safe_div(real_hd["peso_post_g"], real_hd["peso_base_g"])

    # filtro as_of (NO usar hoy)
    real_hd = real_hd[(real_hd["fecha_cosecha"] <= AOD) & (real_hd["fecha_post"] <= AOD)].copy()

    # colapsar real a grano único por (fecha_cosecha, grado, destino)
    # (si hay múltiples destinos/grado por día, sum tallos y ponderar hidratación)
    rg = (
        real_hd.groupby(["fecha_cosecha", "grado", "destino"], as_index=False)
        .agg(
            tallos=("tallos", "sum"),
            dh_dias=("dh_dias", "median"),
            peso_base_g=("peso_base_g", "median"),
            peso_post_g=("peso_post_g", "median"),
            factor_hidr_real=("factor_hidr_real", "median"),
            fecha_post_real=("fecha_post", "median"),
        )
    )

    # =========================
    # Canon real merma/ajuste
    # =========================
    _require_cols(
        real_ma,
        ["fecha_post", "destino", "factor_desp", "factor_ajuste"],
        "dim_mermas_ajuste_fecha_post_destino",
    )
    real_ma["fecha_post"] = _to_date(real_ma["fecha_post"])
    real_ma["destino"] = _canon_str(real_ma["destino"])
    real_ma["factor_desp"] = pd.to_numeric(real_ma["factor_desp"], errors="coerce")
    real_ma["factor_ajuste"] = pd.to_numeric(real_ma["factor_ajuste"], errors="coerce")
    real_ma = real_ma[real_ma["fecha_post"] <= AOD].copy()

    # =========================
    # JOIN: pred vs real (DH/HIDR) por fecha_cosecha (pred.fecha)
    # =========================
    df = pred.merge(
        rg,
        left_on=["fecha", "grado", "destino"],
        right_on=["fecha_cosecha", "grado", "destino"],
        how="left",
    )

    # JOIN merma/ajuste real por fecha_post_real (prefer) y destino.
    # Si no hay fecha_post_real, usar fecha_post_pred (evaluación ?contra calendario?)
    df["fecha_post_key"] = df["fecha_post_real"]
    df["fecha_post_key"] = df["fecha_post_key"].where(df["fecha_post_key"].notna(), df[fecha_post_pred_col])
    df = df.merge(
        real_ma.rename(columns={"fecha_post": "fecha_post_key"}),
        on=["fecha_post_key", "destino"],
        how="left",
        suffixes=("", "_realma"),
    )

    # =========================
    # KPIs por componente
    # =========================
    # Pesos: usar tallos reales como peso
    w = pd.to_numeric(df["tallos"], errors="coerce").fillna(0.0)

    # --- DH ---
    if dh_col:
        df["dh_pred"] = pd.to_numeric(df[dh_col], errors="coerce")
    else:
        df["dh_pred"] = pd.to_numeric(_safe_div((df[fecha_post_pred_col] - df["fecha"]).dt.days, 1.0), errors=
"coerce")

    df["dh_real"] = pd.to_numeric(df["dh_dias"], errors="coerce")
    m_dh = df["dh_pred"].notna() & df["dh_real"].notna()
    df.loc[m_dh, "err_dh_days"] = df.loc[m_dh, "dh_pred"] - df.loc[m_dh, "dh_real"]
    df.loc[m_dh, "abs_err_dh_days"] = df.loc[m_dh, "err_dh_days"].abs()

    # --- HIDR ---
    if hidr_col:
        df["hidr_pred_factor"] = pd.to_numeric(df[hidr_col], errors="coerce")
    else:
        df["hidr_pred_factor"] = np.nan
    df["hidr_real_factor"] = pd.to_numeric(df["factor_hidr_real"], errors="coerce")

    m_h = df["hidr_pred_factor"].notna() & df["hidr_real_factor"].notna()
    df.loc[m_h, "ratio_hidr"] = _safe_div(df.loc[m_h, "hidr_real_factor"], df.loc[m_h, "hidr_pred_factor"])
    df.loc[m_h, "log_ratio_hidr"] = np.log(df.loc[m_h, "ratio_hidr"].clip(lower=1e-6))

    # --- DESP ---
    if desp_col:
        df["desp_pred_factor"] = pd.to_numeric(df[desp_col], errors="coerce")
    else:
        df["desp_pred_factor"] = np.nan
    df["desp_real_factor"] = pd.to_numeric(df["factor_desp"], errors="coerce")

    m_d = df["desp_pred_factor"].notna() & df["desp_real_factor"].notna()
    df.loc[m_d, "ratio_desp"] = _safe_div(df.loc[m_d, "desp_real_factor"], df.loc[m_d, "desp_pred_factor"])
    df.loc[m_d, "log_ratio_desp"] = np.log(df.loc[m_d, "ratio_desp"].clip(lower=1e-6))

    # --- AJUSTE ---
    if aj_col:
        df["aj_pred_factor"] = pd.to_numeric(df[aj_col], errors="coerce")
    else:
        df["aj_pred_factor"] = np.nan
    df["aj_real_factor"] = pd.to_numeric(df["factor_ajuste"], errors="coerce")

    m_a = df["aj_pred_factor"].notna() & df["aj_real_factor"].notna()
    df.loc[m_a, "ratio_ajuste"] = _safe_div(df.loc[m_a, "aj_real_factor"], df.loc[m_a, "aj_pred_factor"])
    df.loc[m_a, "log_ratio_ajuste"] = np.log(df.loc[m_a, "ratio_ajuste"].clip(lower=1e-6))

    # =========================
    # GLOBAL KPIs
    # =========================
    def _kpi_block(sub: pd.DataFrame) -> dict:
        ww = pd.to_numeric(sub["tallos"], errors="coerce").fillna(0.0)

        out = {
            "n_rows_pred": int(len(sub)),
            "n_rows_with_real_hd": int(sub["tallos"].notna().sum()),
            "as_of_date": AOD,
            "created_at": pd.Timestamp(datetime.now()).normalize(),
            "dh_col_used": str(dh_col),
            "fecha_post_pred_col_used": str(fecha_post_pred_col),
            "hidr_col_used": str(hidr_col),
            "desp_col_used": str(desp_col),
            "ajuste_col_used": str(aj_col),
        }

        # DH
        dsub = sub[sub["err_dh_days"].notna()].copy()
        out["n_dh"] = int(len(dsub))
        out["mae_dh_days"] = mae(dsub["err_dh_days"])
        out["bias_dh_days"] = float(np.nanmean(pd.to_numeric(dsub["err_dh_days"], errors="coerce"))) if len(ds
ub) else np.nan
        out["wape_abs_dh_days_wt_tallos"] = wape(dsub["abs_err_dh_days"], ww.loc[dsub.index]) if len(dsub) els
e np.nan

        # Hidr
        hsub = sub[sub["log_ratio_hidr"].notna()].copy()
        out["n_hidr"] = int(len(hsub))
        out["mae_log_ratio_hidr"] = mae(hsub["log_ratio_hidr"])
        out["median_ratio_hidr"] = float(np.nanmedian(pd.to_numeric(hsub["ratio_hidr"], errors="coerce"))) if
len(hsub) else np.nan
        out["p05_ratio_hidr"] = float(np.nanpercentile(pd.to_numeric(hsub["ratio_hidr"], errors="coerce"), 5))
 if len(hsub) else np.nan
        out["p95_ratio_hidr"] = float(np.nanpercentile(pd.to_numeric(hsub["ratio_hidr"], errors="coerce"), 95)
) if len(hsub) else np.nan

        # Desp
        dpsub = sub[sub["log_ratio_desp"].notna()].copy()
        out["n_desp"] = int(len(dpsub))
        out["mae_log_ratio_desp"] = mae(dpsub["log_ratio_desp"])
        out["median_ratio_desp"] = float(np.nanmedian(pd.to_numeric(dpsub["ratio_desp"], errors="coerce"))) if
 len(dpsub) else np.nan

        # Ajuste
        asub = sub[sub["log_ratio_ajuste"].notna()].copy()
        out["n_ajuste"] = int(len(asub))
        out["mae_log_ratio_ajuste"] = mae(asub["log_ratio_ajuste"])
        out["median_ratio_ajuste"] = float(np.nanmedian(pd.to_numeric(asub["ratio_ajuste"], errors="coerce")))
 if len(asub) else np.nan
```

```
        return out

    global_df = pd.DataFrame([_kpi_block(df)])

    # =========================
    # BY DESTINO / BY GRADO
    # =========================
    by_dest = []
    for dest, g in df.groupby("destino", dropna=False):
        row = _kpi_block(g)
        row["destino"] = str(dest)
        by_dest.append(row)
    by_dest_df = pd.DataFrame(by_dest).sort_values("n_rows_pred", ascending=False)

    by_gr = []
    for gr, g in df.groupby("grado", dropna=False):
        row = _kpi_block(g)
        row["grado"] = int(gr) if pd.notna(gr) else None
        by_gr.append(row)
    by_gr_df = pd.DataFrame(by_gr).sort_values("n_rows_pred", ascending=False)

    # =========================
    # COVERAGE / SANITY
    # =========================
    cov = pd.DataFrame([{
        "as_of_date": AOD,
        "rows_pred": int(len(pred)),
        "rows_join_real_hd": int(df["tallos"].notna().sum()),
        "coverage_real_hd": float(df["tallos"].notna().mean()) if len(df) else np.nan,
        "rows_join_real_ma": int(df["factor_desp"].notna().sum()),
        "coverage_real_ma": float(df["factor_desp"].notna().mean()) if len(df) else np.nan,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(global_df, OUT_GLOBAL)
    write_parquet(by_dest_df, OUT_BY_DESTINO)
    write_parquet(by_gr_df, OUT_BY_GRADO)
    write_parquet(cov, OUT_COVERAGE)

    print(f"[OK] Wrote global : {OUT_GLOBAL}")
    print(global_df.to_string(index=False))
    print(f"\n[OK] Wrote destino: {OUT_BY_DESTINO} rows={len(by_dest_df)}")
    print(by_dest_df.head(10).to_string(index=False))
    print(f"\n[OK] Wrote grado  : {OUT_BY_GRADO} rows={len(by_gr_df)}")
    print(by_gr_df.head(10).to_string(index=False))
    print(f"\n[OK] Wrote coverage: {OUT_COVERAGE}")
    print(cov.to_string(index=False))


if __name__ == "__main__":
    main(as_of_date=None)

------------------------------------------------------------
[23/65] FILE: \src\eval\build_kpis_share_grado_ml2.py
------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
```

```
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FACTOR = EVAL / "backtest_factor_ml2_share_grado.parquet"
IN_FINAL = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"
IN_REAL = SILVER / "fact_cosecha_real_grado_dia.parquet"
IN_PRED_ML1 = GOLD / "pred_tallos_grado_dia_ml1_full.parquet"

OUT_GLOBAL = EVAL / "ml2_share_grado_eval_global.parquet"
OUT_BY_GRADO = EVAL / "ml2_share_grado_eval_by_grado.parquet"
OUT_SANITY = EVAL / "ml2_share_grado_eval_sanity.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _resolve_block_base(df: pd.DataFrame) -> pd.DataFrame:
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
        return df
    if "bloque_padre" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_padre"])
        return df
    if "bloque" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque"])
        return df
    raise KeyError("No encuentro columna de bloque: bloque_base / bloque_padre / bloque")


def mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan


def wape_weighted(abs_err: pd.Series, w: pd.Series) -> float:
    abs_err = pd.to_numeric(abs_err, errors="coerce").fillna(0.0)
    w = pd.to_numeric(w, errors="coerce").fillna(0.0)
    denom = float(np.sum(w))
    if denom <= 0:
        return np.nan
    return float(np.sum(abs_err * w) / denom)


def main() -> None:
    fac = read_parquet(IN_FACTOR).copy()
    fin = read_parquet(IN_FINAL).copy()
    real = read_parquet(IN_REAL).copy()
    ml1 = read_parquet(IN_PRED_ML1).copy()


    # --- Alias handling: share_final ---
    if "share_final" not in fac.columns:
        # intentos comunes
        cands = [c for c in fac.columns if ("share" in c.lower()) and ("final" in c.lower())]
        if cands:
            fac = fac.rename(columns={cands[0]: "share_final"})
        else:
            raise KeyError(f"No encuentro share_final en factor. Columnas disponibles: {list(fac.columns)}")

    # Canon
    for df in (fac, fin, ml1):
        df["fecha"] = _to_date(df["fecha"])
        df["bloque_base"] = _canon_str(df["bloque_base"])
        df["grado"] = _canon_str(df["grado"])
        if "variedad_canon" in df.columns:
            df["variedad_canon"] = _canon_str(df["variedad_canon"])

    real["fecha"] = _to_date(real["fecha"])
    real = _resolve_block_base(real)
    real["grado"] = _canon_str(real["grado"])
    real["tallos_real"] = pd.to_numeric(real["tallos_real"], errors="coerce").fillna(0.0)

    # Real share por día/bloque
    rg = real.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"
))
    tot = rg.groupby(["fecha", "bloque_base"], as_index=False).agg(tallos_real_dia=("tallos_real", "sum"))
    rg = rg.merge(tot, on=["fecha", "bloque_base"], how="left")
    rg["share_real"] = np.where(rg["tallos_real_dia"] > 0, rg["tallos_real"] / rg["tallos_real_dia"], np.nan)

    # ML1 share
    ml1_need = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_ml1", "tallos_pred
_ml1_grado_dia"]
    ml1 = ml1[ml1_need].copy()
    ml1["share_grado_ml1"] = pd.to_numeric(ml1["share_grado_ml1"], errors="coerce")
    ml1["tallos_pred_ml1_grado_dia"] = pd.to_numeric(ml1["tallos_pred_ml1_grado_dia"], errors="coerce").fillna
(0.0)

    # Merge everything at grain (ciclo_id, fecha, bloque_base, variedad_canon, grado)
    fac_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]
    opt = []
    if "share_ml1" in fac.columns:
        opt.append("share_ml1")
    elif "share_grado_ml1" in fac.columns:
        fac = fac.rename(columns={"share_grado_ml1": "share_ml1"})
        opt.append("share_ml1")

    # fin YA trae share_final (porque apply lo guarda ahí).
    # del factor solo necesitamos share_ml1 (baseline) para comparar.

    fac_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]

    # normalizamos share_ml1 desde factor
    if "share_ml1" in fac.columns:
        fac_use = fac[fac_cols + ["share_ml1"]].copy()
    elif "share_grado_ml1" in fac.columns:
        fac_use = fac.rename(columns={"share_grado_ml1": "share_ml1"})[fac_cols + ["share_ml1"]].copy()
    else:
        raise KeyError(f"No encuentro share_ml1 / share_grado_ml1 en factor. Columnas: {list(fac.columns)}")

    df = fin.merge(
        fac_use,
        on=["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"],
        how="left",
    )



    df = df.merge(ml1, on=["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"], how="left")
    df = df.merge(rg[["fecha", "bloque_base", "grado", "share_real", "tallos_real", "tallos_real_dia"]],
                  on=["fecha", "bloque_base", "grado"], how="left")

    # Cast
    df["share_final"] = pd.to_numeric(df["share_final"], errors="coerce")
    df["share_grado_ml1"] = pd.to_numeric(df["share_grado_ml1"], errors="coerce")
    df["share_real"] = pd.to_numeric(df["share_real"], errors="coerce")

    df["tallos_final_grado_dia"] = pd.to_numeric(df["tallos_final_grado_dia"], errors="coerce")
    df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce")
    df["tallos_total_ml2"] = pd.to_numeric(df["tallos_total_ml2"], errors="coerce")

    # --- KPI share error ---
    m = df["share_real"].notna() & df["share_grado_ml1"].notna() & df["share_final"].notna()
    d = df.loc[m].copy()

    d["err_share_ml1"] = d["share_real"] - d["share_grado_ml1"]
    d["err_share_ml2"] = d["share_real"] - d["share_final"]
    d["abs_err_share_ml1"] = d["err_share_ml1"].abs()
    d["abs_err_share_ml2"] = d["err_share_ml2"].abs()

    # Weighted by tallos_real_dia (impacto por día)
    w = pd.to_numeric(d["tallos_real_dia"], errors="coerce").fillna(0.0)

    # --- KPI tallos por grado error (impacto directo) ---
    # Error absoluto en tallos (si hay real)
    d["abs_err_tallos_ml1"] = (d["tallos_real"] - d["tallos_pred_ml1_grado_dia"]).abs()
    d["abs_err_tallos_ml2"] = (d["tallos_real"] - d["tallos_final_grado_dia"]).abs()

    # Global
    out_g = pd.DataFrame([{
        "n_rows": int(len(d)),
        "n_cycles": int(d["ciclo_id"].nunique()),
        "mae_share_ml1": mae(d["err_share_ml1"]),
        "mae_share_ml2": mae(d["err_share_ml2"]),
        "wape_share_ml1_wt_day": wape_weighted(d["abs_err_share_ml1"], w),
        "wape_share_ml2_wt_day": wape_weighted(d["abs_err_share_ml2"], w),
        "mae_abs_tallos_ml1": float(np.nanmean(d["abs_err_tallos_ml1"])),
        "mae_abs_tallos_ml2": float(np.nanmean(d["abs_err_tallos_ml2"])),
        "improvement_abs_mae_share": mae(d["err_share_ml1"]) - mae(d["err_share_ml2"]),
        "improvement_abs_mae_abs_tallos": float(np.nanmean(d["abs_err_tallos_ml1"]) - np.nanmean(d["abs_err_ta
llos_ml2"])),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    # By grado
    rows = []
    for grado, g in d.groupby("grado"):
        ww = pd.to_numeric(g["tallos_real_dia"], errors="coerce").fillna(0.0)
        rows.append({
            "grado": str(grado),
            "n_rows": int(len(g)),
            "mae_share_ml1": mae(g["err_share_ml1"]),
            "mae_share_ml2": mae(g["err_share_ml2"]),
            "wape_share_ml1_wt_day": wape_weighted(g["abs_err_share_ml1"], ww),
            "wape_share_ml2_wt_day": wape_weighted(g["abs_err_share_ml2"], ww),
            "mae_abs_tallos_ml1": float(np.nanmean(g["abs_err_tallos_ml1"])),
            "mae_abs_tallos_ml2": float(np.nanmean(g["abs_err_tallos_ml2"])),
            "improvement_abs_mae_share": mae(g["err_share_ml1"]) - mae(g["err_share_ml2"]),
        })
    out_bg = pd.DataFrame(rows).sort_values("improvement_abs_mae_share", ascending=False)

    # --- Sanity: sum shares ---
    s = fac.groupby(["ciclo_id", "fecha"], as_index=False).agg(sum_share_final=("share_final", "sum"))
    sanity = pd.DataFrame([{
        "n_days": int(len(s)),
        "sum_share_min": float(s["sum_share_final"].min()),
        "sum_share_p25": float(s["sum_share_final"].quantile(0.25)),
        "sum_share_median": float(s["sum_share_final"].median()),
        "sum_share_p75": float(s["sum_share_final"].quantile(0.75)),
        "sum_share_max": float(s["sum_share_final"].max()),
        "pct_close_1pm_1e-6": float(np.mean(np.isclose(s["sum_share_final"].values, 1.0, atol=1e-6))),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(out_g, OUT_GLOBAL)
    write_parquet(out_bg, OUT_BY_GRADO)
    write_parquet(sanity, OUT_SANITY)

    print(f"[OK] Wrote global: {OUT_GLOBAL}")
    print(out_g.to_string(index=False))
    print(f"\n[OK] Wrote by_grado: {OUT_BY_GRADO} rows={len(out_bg)}")
    print(out_bg.head(10).to_string(index=False))
    print(f"\n[OK] Wrote sanity: {OUT_SANITY}")
    print(sanity.to_string(index=False))


if __name__ == "__main__":
    main()
```

```
----------------------------------------------------
[24/65] FILE: \src\eval\build_kpis_tallos_curve_ml2.py
----------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_FACTORS = EVAL_DIR / "backtest_factor_ml2_tallos_curve_dia.parquet"
IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"

OUT_GLOBAL = EVAL_DIR / "ml2_tallos_curve_eval_global.parquet"
OUT_GROUP = EVAL_DIR / "ml2_tallos_curve_eval_by_group.parquet"
OUT_RATIO = EVAL_DIR / "ml2_tallos_curve_eval_ratio_dist.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _mae(err: pd.Series) -> float:
    err = pd.to_numeric(err, errors="coerce")
    return float(np.nanmean(np.abs(err))) if len(err) else float("nan")


def _wape(y_true: pd.Series, y_pred: pd.Series) -> float:
    y_true = pd.to_numeric(y_true, errors="coerce").fillna(0.0)
    y_pred = pd.to_numeric(y_pred, errors="coerce").fillna(0.0)
    denom = float(np.sum(np.abs(y_true)))
    if denom <= 0:
        return float("nan")
    return float(np.sum(np.abs(y_true - y_pred)) / denom)


def _smape(y_true: pd.Series, y_pred: pd.Series) -> float:
    y_true = pd.to_numeric(y_true, errors="coerce").fillna(0.0)
    y_pred = pd.to_numeric(y_pred, errors="coerce").fillna(0.0)
    denom = np.abs(y_true) + np.abs(y_pred)
    m = denom > 0
    if not m.any():
        return float("nan")
    return float(np.mean(2.0 * np.abs(y_true[m] - y_pred[m]) / denom[m]))


def main() -> None:
    df = read_parquet(IN_FACTORS).copy()

    # Attach grouping fields if missing (estado / tipo_sp / area)
    ciclo = read_parquet(IN_CICLO).copy()
    if "ciclo_id" not in ciclo.columns:
        raise KeyError("fact_ciclo_maestro debe tener ciclo_id")

    # Normalizar estado si viene
    if "estado" in ciclo.columns:
        ciclo["estado"] = ciclo["estado"].astype(str).str.upper().str.strip()

    ciclo_cols = ["ciclo_id"]
    for c in ["estado", "tipo_sp", "area"]:
        if c in ciclo.columns:
            ciclo_cols.append(c)

    df = df.merge(ciclo[ciclo_cols].drop_duplicates("ciclo_id"), on="ciclo_id", how="left")

    # Core
    df["tallos_real_dia"] = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0)
    df["tallos_pred_ml1_dia"] = pd.to_numeric(df["tallos_pred_ml1_dia"], errors="coerce").fillna(0.0)
    df["tallos_final_ml2_dia"] = pd.to_numeric(df["tallos_final_ml2_dia"], errors="coerce").fillna(0.0)

    # Solo rows con señal (evita universo vacío)
    df = df.loc[(df["tallos_real_dia"] > 0) | (df["tallos_pred_ml1_dia"] > 0), :].copy()

    # Errors
    df["err_ml1"] = df["tallos_real_dia"] - df["tallos_pred_ml1_dia"]
    df["err_ml2"] = df["tallos_real_dia"] - df["tallos_final_ml2_dia"]

    # Global KPIs
    out_g = pd.DataFrame([{
        "n_rows": int(len(df)),
        "n_cycles": int(df["ciclo_id"].nunique()),
        "mae_ml1": _mae(df["err_ml1"]),
        "mae_ml2": _mae(df["err_ml2"]),
        "wape_ml1": _wape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"]),
        "wape_ml2": _wape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
        "smape_ml1": _smape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"]),
        "smape_ml2": _smape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
        "improvement_abs_mae": _mae(df["err_ml1"]) - _mae(df["err_ml2"]),
        "improvement_abs_wape": _wape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"]) - _wape(df["tallos_rea
l_dia"], df["tallos_final_ml2_dia"]),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    # Ratio distribution sanity
    r = pd.to_numeric(df["pred_ratio"], errors="coerce")
    out_r = pd.DataFrame([{
        "n": int(r.notna().sum()),
        "ratio_min": float(np.nanmin(r)) if len(r) else np.nan,
        "ratio_p05": float(np.nanpercentile(r, 5)) if len(r) else np.nan,
        "ratio_p25": float(np.nanpercentile(r, 25)) if len(r) else np.nan,
        "ratio_median": float(np.nanmedian(r)) if len(r) else np.nan,
        "ratio_p75": float(np.nanpercentile(r, 75)) if len(r) else np.nan,
        "ratio_p95": float(np.nanpercentile(r, 95)) if len(r) else np.nan,
        "ratio_max": float(np.nanmax(r)) if len(r) else np.nan,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])

    # Grouping: prefer estado, else tipo_sp, else area
    if "estado" in df.columns and df["estado"].notna().any():
        group_col = "estado"
    elif "tipo_sp" in df.columns and df["tipo_sp"].notna().any():
        group_col = "tipo_sp"
    else:
        group_col = "area" if "area" in df.columns else None

    rows = []
    if group_col:
        for k, g in df.groupby(group_col):
            rows.append({
                "group_col": group_col,
                "group_val": str(k),
                "n_rows": int(len(g)),
                "n_cycles": int(g["ciclo_id"].nunique()),
                "mae_ml1": _mae(g["err_ml1"]),
                "mae_ml2": _mae(g["err_ml2"]),
                "wape_ml1": _wape(g["tallos_real_dia"], g["tallos_pred_ml1_dia"]),
                "wape_ml2": _wape(g["tallos_real_dia"], g["tallos_final_ml2_dia"]),
                "improvement_abs_mae": _mae(g["err_ml1"]) - _mae(g["err_ml2"]),
                "improvement_abs_wape": _wape(g["tallos_real_dia"], g["tallos_pred_ml1_dia"]) - _wape(g["tallo
s_real_dia"], g["tallos_final_ml2_dia"]),
            })
    out_grp = pd.DataFrame(rows)
    # Write
    EVAL_DIR.mkdir(parents=True, exist_ok=True)
    write_parquet(out_g, OUT_GLOBAL)
    write_parquet(out_grp, OUT_GROUP)
    write_parquet(out_r, OUT_RATIO)

    print(f"[OK] Wrote global: {OUT_GLOBAL}")
    print(out_g.to_string(index=False))
    print(f"\n[OK] Wrote group : {OUT_GROUP} (group_col={group_col}) rows={len(out_grp)}")
    if not out_grp.empty:
        print(out_grp.sort_values("improvement_abs_wape", ascending=False).head(15).to_string(index=False))
    print(f"\n[OK] Wrote ratio : {OUT_RATIO}")
    print(out_r.to_string(index=False))


if __name__ == "__main__":
    main()

------------------------------------------------------------
[25/65] FILE: \src\gold\build_ds_ajuste_poscosecha_ml2_v1.py
------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"
IN_REAL_MA = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

OUT_DS = GOLD / "ml2_datasets" / "ds_ajuste_poscosecha_ml2_v1.parquet"


# ----------------------------
# TZ helpers (evita tz-aware vs tz-naive)
# ----------------------------
def _to_naive_utc(ts: pd.Timestamp) -> pd.Timestamp:
    """Convierte a UTC y deja tz-naive."""
    if ts.tzinfo is None:
        return ts
    return ts.tz_convert("UTC").tz_localize(None)


def _as_of_date_naive() -> pd.Timestamp:
    """
    Regla: as_of = hoy-1.
    IMPORTANTE: tz-naive para comparar con columnas datetime64[ns] (naive).
    """
    # utcnow() suele ser tz-naive, pero en algunos entornos puede salir tz-aware; blindamos.
    t = pd.Timestamp.utcnow()
    t = _to_naive_utc(t)
    return (t.normalize() - pd.Timedelta(days=1))


def _to_date_naive(s: pd.Series) -> pd.Series:
    """
    Convierte a datetime y deja tz-naive normalizado.
    """
    dt = pd.to_datetime(s, errors="coerce")
    # Si viene tz-aware, pásalo a UTC y quita tz
    try:
        if getattr(dt.dt, "tz", None) is not None:
            dt = dt.dt.tz_convert("UTC").dt.tz_localize(None)
    except Exception:
        # si no soporta .dt.tz (por tipos raros), lo dejamos como está
        pass
    return dt.dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:
        if c in df.columns:
            return c
    raise KeyError(
        "No encuentro columna de fecha_post_pred (esperaba fecha_post_pred_final/fecha_post_pred_ml1/...)."
    )


def _resolve_ajuste_ml1(df: pd.DataFrame) -> str:
    for c in ["factor_ajuste_ml1", "ajuste_ml1", "factor_ajuste_seed", "factor_ajuste"]:
        if c in df.columns:
            return c
    raise KeyError(
        "No encuentro columna de ajuste ML1 (esperaba ajuste_ml1 / factor_ajuste_ml1 / factor_ajuste_seed)."
    )


def _weight_series(df: pd.DataFrame) -> pd.Series:
    for c in ["tallos_w", "tallos", "tallos_total_ml2", "tallos_total"]:
        if c in df.columns:
            return pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    return pd.Series(1.0, index=df.index, dtype="float64")


def main() -> None:
    as_of = _as_of_date_naive()

    uni = read_parquet(IN_UNIVERSE).copy()
    uni.columns = [str(c).strip() for c in uni.columns]

    need = {"fecha", "destino"}
    miss = need - set(uni.columns)
    if miss:
        raise ValueError(f"Universe sin columnas: {sorted(miss)}")

    # ? tz-naive
    uni["fecha"] = _to_date_naive(uni["fecha"])
    uni["destino"] = _canon_str(uni["destino"])

    if "grado" in uni.columns:
        uni["grado"] = pd.to_numeric(uni["grado"], errors="coerce").astype("Int64")
    else:
        uni["grado"] = pd.Series(pd.NA, index=uni.index, dtype="Int64")

    fecha_post_col = _resolve_fecha_post_pred(uni)
    uni[fecha_post_col] = _to_date_naive(uni[fecha_post_col])

    ajuste_ml1_col = _resolve_ajuste_ml1(uni)
    uni[ajuste_ml1_col] = pd.to_numeric(uni[ajuste_ml1_col], errors="coerce")

    # ? comparación safe: ambos tz-naive
    uni = uni[uni["fecha"].notna() & (uni["fecha"] <= as_of)].copy()

    # ---- Real por fecha_post + destino (NO por grado) ----
    real = read_parquet(IN_REAL_MA).copy()
    real.columns = [str(c).strip() for c in real.columns]
```

```
if "fecha_post" not in real.columns or "destino" not in real.columns:
    raise ValueError("dim_mermas_ajuste_fecha_post_destino debe traer fecha_post y destino.")

if "factor_ajuste" in real.columns:
    real_factor_col = "factor_ajuste"
elif "ajuste" in real.columns:
    real_factor_col = "ajuste"
else:
    raise ValueError("dim_mermas_ajuste_fecha_post_destino no trae factor_ajuste ni ajuste.")

real["fecha_post"] = _to_date_naive(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
real[real_factor_col] = pd.to_numeric(real[real_factor_col], errors="coerce")

real2 = (
    real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
        .agg(factor_ajuste_real=(real_factor_col, "median"))
)

df = uni.merge(
    real2.rename(columns={"fecha_post": "fecha_post_key"}),
    left_on=[fecha_post_col, "destino"],
    right_on=["fecha_post_key", "destino"],
    how="left",
)

# target: log(real / ml1)  (si es >0)
ml1 = pd.to_numeric(df[ajuste_ml1_col], errors="coerce")
realv = pd.to_numeric(df["factor_ajuste_real"], errors="coerce")

ratio = realv / ml1.replace(0, np.nan)
log_ratio = np.log(ratio.replace([np.inf, -np.inf], np.nan))

clip = 1.2
df["log_ratio_ajuste"] = log_ratio.clip(lower=-clip, upper=clip)

df["w"] = _weight_series(df)

# calendario sobre fecha_post_pred
df["dow"] = df[fecha_post_col].dt.dayofweek.astype("Int64")
df["month"] = df[fecha_post_col].dt.month.astype("Int64")
df["weekofyear"] = df[fecha_post_col].dt.isocalendar().week.astype("Int64")

keep = [
    "fecha",
    fecha_post_col,
    "bloque_base" if "bloque_base" in df.columns else None,
    "variedad_canon" if "variedad_canon" in df.columns else None,
    "grado",
    "destino",
    ajuste_ml1_col,
    "factor_ajuste_real",
    "log_ratio_ajuste",
    "w",
    "dow",
    "month",
    "weekofyear",
]
keep = [c for c in keep if c is not None and c in df.columns]

out = df[keep].copy()
out = out.rename(columns={fecha_post_col: "fecha_post_pred_used", ajuste_ml1_col: "ajuste_ml1_used"})
out["as_of_date"] = as_of
out["created_at"] = pd.Timestamp.utcnow()

(GOLD / "ml2_datasets").mkdir(parents=True, exist_ok=True)
write_parquet(out, OUT_DS)

n_real = int(out["factor_ajuste_real"].notna().sum())
print(f"[OK] Wrote dataset: {OUT_DS}")
print(f"     rows={len(out):,} rows_with_real={n_real:,} as_of_date={as_of.date()} clip=±{clip}")
```

```
if __name__ == "__main__":
    main()

----------------------------------------------------------
[26/65] FILE: \src\gold\build_ds_desp_poscosecha_ml2_v1.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

# Universe: salida final de HIDR (ya tiene fecha_post_pred usada y ML1 hidr aplicado)
IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"

# Real desperdicio (por fecha_post, destino)
IN_REAL = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

OUT_DS = GOLD / "ml2_datasets" / "ds_desp_poscosecha_ml2_v1.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _as_of_date(default_tz: str = "America/Guayaquil") -> pd.Timestamp:
    # ?hoy? no se usa (día no cerrado) => as_of_date = hoy-1
    # pd.Timestamp.now() toma tz local del OS; usamos normalize y -1 día.
    return (pd.Timestamp.now().normalize() - pd.Timedelta(days=1))


def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    for c in [
        "fecha_post_pred_final",
        "fecha_post_pred_used",
        "fecha_post_pred_ml1",
        "fecha_post_pred",
    ]:
        if c in df.columns:
            return c
    raise KeyError(
        "No encuentro columna de fecha_post_pred. Espero una de: "
        "fecha_post_pred_final / fecha_post_pred_used / fecha_post_pred_ml1 / fecha_post_pred"
    )


def _resolve_factor_desp_ml1(df: pd.DataFrame) -> str:
    for c in ["factor_desp_ml1", "factor_desp_pred_ml1", "factor_desp"]:
        if c in df.columns:
            return c
    raise KeyError(
        "No encuentro factor_desp ML1 en universe. Espero: factor_desp_ml1 / factor_desp_pred_ml1 / factor_des
p"
    )


def _resolve_tallos(df: pd.DataFrame) -> str:
    for c in [
        "tallos",
        "tallos_total_ml2",
        "tallos_real_dia",
        "tallos_pred_ml1_grado_dia",
        "tallos_final_grado_dia",
        "tallos_pred_ml1_grado_dia",
    ]:
        if c in df.columns:
            return c
    # si no hay tallos, usamos 1.0 como peso
    return ""


def main() -> None:
    as_of = _as_of_date()

    uni = read_parquet(IN_UNIVERSE).copy()
    uni.columns = [str(c).strip() for c in uni.columns]

    # Canon mínimas
    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
    miss = need - set(uni.columns)
    if miss:
        raise ValueError(f"Universe sin columnas: {sorted(miss)}")

    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_str(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])
    uni["destino"] = _canon_str(uni["destino"])
    # grado a str/cat
    if pd.api.types.is_numeric_dtype(uni["grado"]):
        uni["grado"] = _canon_int(uni["grado"]).astype("Int64")
    else:
        uni["grado"] = _canon_str(uni["grado"])

    # Cortar al as_of_date
    uni = uni.loc[uni["fecha"].notna() & (uni["fecha"] <= as_of)].copy()

    fpp_col = _resolve_fecha_post_pred(uni)
    fd_ml1_col = _resolve_factor_desp_ml1(uni)

    uni[fpp_col] = _to_date(uni[fpp_col])
    uni[fd_ml1_col] = pd.to_numeric(uni[fd_ml1_col], errors="coerce")

    tallos_col = _resolve_tallos(uni)
    if tallos_col:
        uni["tallos_w"] = pd.to_numeric(uni[tallos_col], errors="coerce").fillna(0.0)
    else:
        uni["tallos_w"] = 1.0

    # REAL: desperdicio por fecha_post + destino
    real = read_parquet(IN_REAL).copy()
    real.columns = [str(c).strip() for c in real.columns]
    need_r = {"fecha_post", "destino", "factor_desp"}
    miss_r = need_r - set(real.columns)
    if miss_r:
        raise ValueError(f"Real desperdicio sin columnas: {sorted(miss_r)}")

    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    real["factor_desp_real"] = pd.to_numeric(real["factor_desp"], errors="coerce")

    # Colapsar a 1 fila por (fecha_post, destino) usando mediana (robusto)
    real2 = (
        real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
        .agg(factor_desp_real=("factor_desp_real", "median"))
    )

    # Join: fecha_post_pred + destino
    ds = uni.merge(
        real2,
        left_on=[fpp_col, "destino"],
        right_on=["fecha_post", "destino"],
        how="left",
    )

    ds["factor_desp_ml1"] = pd.to_numeric(ds[fd_ml1_col], errors="coerce")
    ds["factor_desp_real"] = pd.to_numeric(ds["factor_desp_real"], errors="coerce")

    # Target: log_ratio_desp = log(real/ml1)
    eps = 1e-12
    ratio = (ds["factor_desp_real"] + eps) / (ds["factor_desp_ml1"] + eps)
    ds["log_ratio_desp"] = np.log(ratio)

    # Clip target (para estabilidad)
    CLIP = 1.2
    ds["log_ratio_desp_clipped"] = ds["log_ratio_desp"].clip(lower=-CLIP, upper=CLIP)

    # Features calendario sobre fecha_post_pred usada
    ds["fecha_post_pred_used"] = ds[fpp_col]
    ds["dow"] = ds["fecha_post_pred_used"].dt.dayofweek.astype("Int64")
    ds["month"] = ds["fecha_post_pred_used"].dt.month.astype("Int64")
    ds["weekofyear"] = ds["fecha_post_pred_used"].dt.isocalendar().week.astype("Int64")

    # Flags de disponibilidad real
    ds["has_real"] = ds["factor_desp_real"].notna()

    ds["as_of_date"] = as_of
    ds["created_at"] = pd.Timestamp(datetime.now()).normalize()

    OUT_DS.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(ds, OUT_DS)

    print(f"[OK] Wrote dataset: {OUT_DS}")
    print(
        f"     rows={len(ds):,} rows_with_real={int(ds['has_real'].sum()):,} "
        f"as_of_date={as_of.date()} clip=±{CLIP}"
    )


if __name__ == "__main__":
    main()

--------------------------------------------------------
[27/65] FILE: \src\gold\build_ds_dh_poscosecha_ml2_v1.py
--------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime, timedelta

import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"
```

```
# Universe: ya contiene el split por destino y llaves bloque/variedad/grado/destino
IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_seed_grado_dia_bloque_destino.parquet"

# DH baseline aplicado (ML1) sobre seed ML2 (esto te da dh_dias_ml1 y fecha_post_pred_ml1)
IN_DH_ML1_ON_ML2 = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino.parquet"

# Real DH + hidratación (tiene dh_dias a nivel fecha_cosecha, fecha_post, grado, destino)
IN_REAL_HD = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_DS = GOLD / "ml2_datasets" / "ds_dh_poscosecha_ml2_v1.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _as_of_date() -> pd.Timestamp:
    # Regla: usar hoy-1 (día cerrado)
    return pd.Timestamp.now().normalize() - pd.Timedelta(days=1)


def main(clip_err_days: float = 5.0) -> None:
    as_of = _as_of_date()

    uni = read_parquet(IN_UNIVERSE).copy()
    dhm = read_parquet(IN_DH_ML1_ON_ML2).copy()
    real = read_parquet(IN_REAL_HD).copy()

    for df in (uni, dhm, real):
        df.columns = [str(c).strip() for c in df.columns]

    # Canon keys
    uni["fecha"] = _to_date(uni["fecha"])
    uni["destino"] = _canon_str(uni["destino"])
    uni["grado"] = _canon_int(uni["grado"])
    uni["bloque_base"] = _canon_int(uni.get("bloque_base", pd.Series([pd.NA] * len(uni))))
    if "variedad_canon" in uni.columns:
        uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    # Filtrar as_of (no usar hoy)
    uni = uni[uni["fecha"].notna() & (uni["fecha"] <= as_of)].copy()

    # DH ML1 sobre ML2 seed
    dhm["fecha"] = _to_date(dhm["fecha"])
    dhm["destino"] = _canon_str(dhm["destino"])
    dhm["grado"] = _canon_int(dhm["grado"])
    dhm["bloque_base"] = _canon_int(dhm.get("bloque_base", pd.Series([pd.NA] * len(dhm))))
    if "variedad_canon" in dhm.columns:
        dhm["variedad_canon"] = _canon_str(dhm["variedad_canon"])

    # Identificar columna DH ML1
    dh_col = None
    for c in ["dh_dias_ml1", "dh_dias_pred_ml1", "dh_dias_pred", "dh_dias"]:
        if c in dhm.columns:
            dh_col = c
            break
    if dh_col is None:
        raise KeyError("No encuentro columna dh en pred_poscosecha_ml2_dh_*. Espero dh_dias_ml1 o similar.")

    dhm[dh_col] = _canon_int(dhm[dh_col])

    # Real
    real["fecha_cosecha"] = _to_date(real["fecha_cosecha"])
    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    real["grado"] = _canon_int(real["grado"])

    # tallos real (para peso del KPI / entrenamiento)
    if "tallos" in real.columns:
        real["tallos"] = pd.to_numeric(real["tallos"], errors="coerce").fillna(0.0)
    else:
        real["tallos"] = 0.0

    # dh real (col explícita)
    if "dh_dias" in real.columns:
        real["dh_dias"] = _canon_int(real["dh_dias"])
    else:
        # fallback: (fecha_post - fecha_cosecha)
        real["dh_dias"] = (real["fecha_post"] - real["fecha_cosecha"]).dt.days.astype("Int64")

    # Agregar real a grano (fecha_cosecha, grado, destino) con median para dh y sum tallos
    real_g = (
        real.groupby(["fecha_cosecha", "grado", "destino"], dropna=False, as_index=False)
            .agg(
                dh_real=("dh_dias", "median"),
                tallos_real=("tallos", "sum"),
                fecha_post_real=("fecha_post", "min"),
            )
    )

    # Unimos universe + dh_ml1
    keys = ["fecha", "bloque_base", "variedad_canon", "grado", "destino"]
    need_uni = [c for c in keys if c in uni.columns]
    need_dhm = [c for c in keys if c in dhm.columns]

    df = uni.merge(
        dhm[need_dhm + [dh_col]].rename(columns={dh_col: "dh_ml1"}),
        on=[c for c in keys if c in need_uni and c in need_dhm],
        how="left",
    )

    # Join real por fecha_cosecha=fecha + grado + destino
    df = df.merge(
        real_g,
        left_on=["fecha", "grado", "destino"],
        right_on=["fecha_cosecha", "grado", "destino"],
        how="left",
    )

    # Features calendario sobre fecha_cosecha (fecha)
    df["dow"] = df["fecha"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype("Int64")

    # Target: error en días
    df["dh_ml1"] = _canon_int(df["dh_ml1"])
    df["dh_real"] = _canon_int(df["dh_real"])

    df["err_dh_days"] = (df["dh_real"].astype("float") - df["dh_ml1"].astype("float"))
    df["err_dh_days_clipped"] = df["err_dh_days"].clip(lower=-float(clip_err_days), upper=float(clip_err_days)
)

    df["clip_err_days"] = float(clip_err_days)
    df["as_of_date"] = as_of
    df["created_at"] = pd.Timestamp.utcnow()

    # Guardar DS completo (incluye filas sin real; el train filtrará)
    OUT_DS.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(df, OUT_DS)

    n_all = len(df)
    n_real = int(df["dh_real"].notna().sum())
    print(f"[OK] Wrote dataset: {OUT_DS}")
    print(f"     rows={n_all:,} rows_with_real={n_real:,} as_of_date={as_of.date()} clip=±{clip_err_days:g}")
```

```
if __name__ == "__main__":
    main()

----------------------------------------------------------
[28/65] FILE: \src\gold\build_ds_harvest_horizon_ml2_v2.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    # .../src/gold/file.py -> repo_root = parents[2]
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_GRID_ML1 = GOLD_DIR / "universe_harvest_grid_ml1.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"
IN_FACTOR_SOH = GOLD_DIR / "factors" / "factor_ml2_harvest_start.parquet"  # ML2 Inicio (prod)

OUT_DS = GOLD_DIR / "ml2_datasets" / "ds_harvest_horizon_ml2_v2.parquet"

# As-of sampling (V2)
ASOF_FREQ_DAYS = 7
MIN_DAYS_AFTER_SP = 14
MAX_ASOF_POINTS_PER_CICLO = 10

# Real duration sanity
MIN_REAL_DAYS = 1
MAX_REAL_DAYS = 120


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    b = b.replace(0, np.nan)
    return (a / b).fillna(0.0)


def _build_cycle_header(grid: pd.DataFrame) -> pd.DataFrame:
    """
    Reduce universe_harvest_grid_ml1 (diario) a 1 fila por ciclo_id.
    """
    g = grid.copy()
    g["harvest_start_pred"] = _to_date(g["harvest_start_pred"])
    g["harvest_end_pred"] = _to_date(g["harvest_end_pred"])
    g["fecha_sp"] = _to_date(g["fecha_sp"])

    head = (
        g.groupby("ciclo_id", as_index=False)
        .agg(
            bloque_base=("bloque_base", "first"),
            variedad_canon=("variedad_canon", "first"),
            area=("area", "first"),
            tipo_sp=("tipo_sp", "first"),
            estado=("estado", "first"),
            tallos_proy=("tallos_proy", "first"),
            fecha_sp=("fecha_sp", "first"),
            harvest_start_pred=("harvest_start_pred", "min"),
            harvest_end_pred=("harvest_end_pred", "max"),
            n_harvest_days_pred=("n_harvest_days_pred", "max"),
            ml1_version=("ml1_version", "max"),
        )
    )
    head["days_sp_to_start_pred"] = (head["harvest_start_pred"] - head["fecha_sp"]).dt.days
    return head


def _make_asof_dates(fecha_sp: pd.Timestamp, inicio_real: pd.Timestamp) -> list[pd.Timestamp]:
    """
    V2: as_of cada ASOF_FREQ_DAYS desde (sp + MIN_DAYS_AFTER_SP) hasta (inicio_real - 1).
    Cap MAX_ASOF_POINTS_PER_CICLO y asegura incluir el último as_of.
    """
    if pd.isna(fecha_sp) or pd.isna(inicio_real):
        return []

    last = inicio_real - pd.Timedelta(days=1)
    start = fecha_sp + pd.Timedelta(days=MIN_DAYS_AFTER_SP)

    if last < start:
        return [max(fecha_sp, last)]

    dates = list(pd.date_range(start=start, end=last, freq=f"{ASOF_FREQ_DAYS}D"))
    if not dates or dates[-1] != last:
        dates.append(last)

    if len(dates) > MAX_ASOF_POINTS_PER_CICLO:
        dates = dates[-MAX_ASOF_POINTS_PER_CICLO:]

    return [pd.Timestamp(d).normalize() for d in dates]


def _features_asof(clima: pd.DataFrame, ciclo_chunk: pd.DataFrame, as_of_date: pd.Timestamp) -> pd.DataFrame:
    """
    Features clima por ciclo_id usando rango [fecha_sp, as_of_date].
    Si un ciclo no tiene clima en el rango, queda con ceros (no se pierde).
    """
    base = ciclo_chunk[["ciclo_id", "bloque_base", "fecha_sp"]].drop_duplicates().copy()
    base["bloque_base"] = _canon_str(base["bloque_base"])
    base["fecha_sp"] = _to_date(base["fecha_sp"])

    cl = clima.copy()
    cl["fecha"] = _to_date(cl["fecha"])
    cl["bloque_base"] = _canon_str(cl["bloque_base"])
    cl = cl.loc[cl["fecha"] <= as_of_date, :].copy()

    m = base.merge(cl, on="bloque_base", how="left")
    m = m.loc[(m["fecha"] >= m["fecha_sp"]) & (m["fecha"] <= as_of_date), :].copy()

    # no hay datos -> ceros
    if m.empty:
        out = base[["ciclo_id"]].copy()
        for c in [
            "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
            "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
            "solar_cum_sp", "solar_7d",
            "temp_avg_7d",
        ]:
            out[c] = 0.0
        return out

    # usar gdc_dia acumulado (ignorar gdc_base)
    m["gdc_dia"] = pd.to_numeric(m["gdc_dia"], errors="coerce").fillna(0.0)
    m["rainfall_mm_dia"] = pd.to_numeric(m["rainfall_mm_dia"], errors="coerce").fillna(0.0)
    m["solar_energy_j_m2_dia"] = pd.to_numeric(m["solar_energy_j_m2_dia"], errors="coerce").fillna(0.0)
    m["temp_avg_dia"] = pd.to_numeric(m["temp_avg_dia"], errors="coerce")
    m["en_lluvia_dia"] = pd.to_numeric(m["en_lluvia_dia"], errors="coerce").fillna(0.0)
    m = m.sort_values(["ciclo_id", "fecha"])

    def _roll_sum(s: pd.Series, w: int) -> pd.Series:
        return s.rolling(window=w, min_periods=1).sum()

    def _roll_mean(s: pd.Series, w: int) -> pd.Series:
        return s.rolling(window=w, min_periods=1).mean()

    m["gdc_7d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 7))
    m["gdc_14d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 14))
    m["rain_7d"] = m.groupby("ciclo_id")["rainfall_mm_dia"].transform(lambda s: _roll_sum(s, 7))
    m["solar_7d"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].transform(lambda s: _roll_sum(s, 7))
    m["enlluvia_days_7d"] = m.groupby("ciclo_id")["en_lluvia_dia"].transform(lambda s: _roll_sum(s, 7))
    m["temp_avg_7d"] = m.groupby("ciclo_id")["temp_avg_dia"].transform(lambda s: _roll_mean(s, 7))

    m["gdc_cum_sp"] = m.groupby("ciclo_id")["gdc_dia"].cumsum()
    m["rain_cum_sp"] = m.groupby("ciclo_id")["rainfall_mm_dia"].cumsum()
    m["solar_cum_sp"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].cumsum()

    last = m.groupby("ciclo_id", as_index=False).tail(1).copy()
    last["days_from_sp"] = (as_of_date - last["fecha_sp"]).dt.days
    last["days_from_sp"] = pd.to_numeric(last["days_from_sp"], errors="coerce").fillna(0).clip(lower=0)
    last["gdc_per_day"] = _safe_div(last["gdc_cum_sp"], last["days_from_sp"].replace(0, np.nan))

    feat_cols = [
        "ciclo_id",
        "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
        "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
        "solar_cum_sp", "solar_7d",
        "temp_avg_7d",
    ]
    last = last[feat_cols].copy()

    out = base[["ciclo_id"]].merge(last, on="ciclo_id", how="left")

    for c in [
        "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
        "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
        "solar_cum_sp", "solar_7d",
    ]:
        out[c] = pd.to_numeric(out[c], errors="coerce").fillna(0.0)

    out["temp_avg_7d"] = pd.to_numeric(out["temp_avg_7d"], errors="coerce")
    med = out["temp_avg_7d"].median()
    if pd.isna(med):
        med = 0.0
    out["temp_avg_7d"] = out["temp_avg_7d"].fillna(med)

    return out


def main() -> None:
    ciclo = read_parquet(IN_CICLO).copy()
    grid = read_parquet(IN_GRID_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    # ML2 inicio (factor) es opcional: si existe lo usamos; si no, fallback a 0
    factor_soh = None
    if IN_FACTOR_SOH.exists():
        factor_soh = read_parquet(IN_FACTOR_SOH).copy()

    # canon y fechas
    ciclo["bloque_base"] = _canon_str(ciclo["bloque_base"])
    ciclo["fecha_sp"] = _to_date(ciclo["fecha_sp"])
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])
    ciclo["fecha_fin_cosecha"] = _to_date(ciclo["fecha_fin_cosecha"])
    ciclo["estado"] = _canon_str(ciclo["estado"])

    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])
    head = _build_cycle_header(grid)

    # Base ciclo
    df0 = ciclo.merge(head, on="ciclo_id", how="inner", suffixes=("", "_ml1"))

    # Requiere reales válidos de inicio/fin + pred n_harvest_days
    df0 = df0.loc[
        df0["fecha_inicio_cosecha"].notna()
        & df0["fecha_fin_cosecha"].notna()
        & df0["n_harvest_days_pred"].notna(),
        :
    ].copy()

    # Real duration
    df0["n_harvest_days_real"] = (df0["fecha_fin_cosecha"] - df0["fecha_inicio_cosecha"]).dt.days + 1
    df0["n_harvest_days_real"] = pd.to_numeric(df0["n_harvest_days_real"], errors="coerce")

    df0 = df0.loc[
        df0["n_harvest_days_real"].between(MIN_REAL_DAYS, MAX_REAL_DAYS, inclusive="both"),
        :
    ].copy()

    df0["n_harvest_days_pred"] = pd.to_numeric(df0["n_harvest_days_pred"], errors="coerce")
    df0 = df0.loc[df0["n_harvest_days_pred"].notna(), :].copy()

    # Target
    df0["error_horizon_days"] = (df0["n_harvest_days_real"] - df0["n_harvest_days_pred"]).astype(int)

    # Panel as_of por ciclo (V2)
    rows = []
    for r in df0.itertuples(index=False):
        for a in _make_asof_dates(r.fecha_sp, r.fecha_inicio_cosecha):
            rows.append((r.ciclo_id, a))
    asof_df = pd.DataFrame(rows, columns=["ciclo_id", "as_of_date"])

    df = df0.merge(asof_df, on="ciclo_id", how="inner")
    df = df.loc[df["as_of_date"] < df["fecha_inicio_cosecha"], :].copy()

    # Calendario
    df["dow"] = df["as_of_date"].dt.dayofweek
    df["month"] = df["as_of_date"].dt.month
    df["weekofyear"] = df["as_of_date"].dt.isocalendar().week.astype(int)
    df["days_from_sp"] = (df["as_of_date"] - df["fecha_sp"]).dt.days

    # Agregar pred_error_start_days (ML2 Inicio) si existe
    df["pred_error_start_days"] = 0.0
    if factor_soh is not None and len(factor_soh):
        factor_soh = factor_soh.copy()
        # si factor tiene as_of_date o no: en prod normalmente es único; aquí usamos el valor por ciclo
        factor_soh["pred_error_start_days"] = pd.to_numeric(factor_soh["pred_error_start_days"], errors="coerc
e")
        soh = factor_soh[["ciclo_id", "pred_error_start_days"]].drop_duplicates("ciclo_id")
        df = df.drop(columns=["pred_error_start_days"]).merge(soh, on="ciclo_id", how="left")
        df["pred_error_start_days"] = pd.to_numeric(df["pred_error_start_days"], errors="coerce").fillna(0.0)

    # Features clima por as_of_date (batch)
    feats = []
    for as_of_date, chunk in df.groupby("as_of_date"):
        feats.append(_features_asof(clima, chunk[["ciclo_id", "bloque_base", "fecha_sp"]], pd.Timestamp(as_of_
date)))
    feat = pd.concat(feats, ignore_index=True) if feats else pd.DataFrame(columns=["ciclo_id"])

    df = df.merge(feat, on="ciclo_id", how="left")

    # Final imputaciones numéricas
    for c in [
        "days_sp_to_start_pred",
        "n_harvest_days_pred",
        "tallos_proy",
        "days_from_sp",
        "pred_error_start_days",
        "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
        "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
        "solar_cum_sp", "solar_7d",
        "temp_avg_7d",
    ]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

    df["created_at"] = pd.Timestamp(datetime.now()).normalize()

    OUT_DS.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(df, OUT_DS)

    print(f"[OK] Wrote dataset: {OUT_DS}")
    print(f"     rows={len(df):,} cycles={df['ciclo_id'].nunique():,} as_of_dates={df['as_of_date'].nunique():
,}")


if __name__ == "__main__":
    main()

--------------------------------------------------------
[29/65] FILE: \src\gold\build_ds_harvest_start_ml2_v2.py
--------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    # .../src/gold/file.py -> repo_root = parents[2]
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_GRID_ML1 = GOLD_DIR / "universe_harvest_grid_ml1.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"

OUT_DS = GOLD_DIR / "ml2_datasets" / "ds_harvest_start_ml2_v2.parquet"

# V2 params
ASOF_FREQ_DAYS = 7
MIN_DAYS_AFTER_SP = 14
MAX_ASOF_POINTS_PER_CICLO = 10


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    b = b.replace(0, np.nan)
    return (a / b).fillna(0.0)


def _build_cycle_header(grid: pd.DataFrame) -> pd.DataFrame:
    """
    Reduce universe_harvest_grid_ml1 (diario) a 1 fila por ciclo_id.
    """
    g = grid.copy()
    g["harvest_start_pred"] = _to_date(g["harvest_start_pred"])
    g["harvest_end_pred"] = _to_date(g["harvest_end_pred"])
    g["fecha_sp"] = _to_date(g["fecha_sp"])

    head = (
        g.groupby("ciclo_id", as_index=False)
        .agg(
            bloque_base=("bloque_base", "first"),
            variedad_canon=("variedad_canon", "first"),
            area=("area", "first"),
            tipo_sp=("tipo_sp", "first"),
            estado=("estado", "first"),
            tallos_proy=("tallos_proy", "first"),
            fecha_sp=("fecha_sp", "first"),              # ? FIX: incluir fecha_sp
            harvest_start_pred=("harvest_start_pred", "min"),
            harvest_end_pred=("harvest_end_pred", "max"),
            n_harvest_days_pred=("n_harvest_days_pred", "max"),
            ml1_version=("ml1_version", "max"),
        )
    )

    # Robustez: si falta alguna fecha, queda NaN
    head["days_sp_to_start_pred"] = (head["harvest_start_pred"] - head["fecha_sp"]).dt.days
    return head



def _make_asof_dates(fecha_sp: pd.Timestamp, inicio_real: pd.Timestamp) -> list[pd.Timestamp]:
    if pd.isna(fecha_sp) or pd.isna(inicio_real):
        return []

    last = inicio_real - pd.Timedelta(days=1)
    start = fecha_sp + pd.Timedelta(days=MIN_DAYS_AFTER_SP)

    if last < start:
        return [max(fecha_sp, last)]

    dates = list(pd.date_range(start=start, end=last, freq=f"{ASOF_FREQ_DAYS}D"))
    if not dates or dates[-1] != last:
        dates.append(last)

    if len(dates) > MAX_ASOF_POINTS_PER_CICLO:
        dates = dates[-MAX_ASOF_POINTS_PER_CICLO:]

    return [pd.Timestamp(d).normalize() for d in dates]


def _features_asof(clima: pd.DataFrame, ciclo_chunk: pd.DataFrame, as_of_date: pd.Timestamp) -> pd.DataFrame:
    cl = clima.copy()
    cl["fecha"] = _to_date(cl["fecha"])
    cl["bloque_base"] = _canon_str(cl["bloque_base"])
    cl = cl.loc[cl["fecha"] <= as_of_date, :].copy()

    cy = ciclo_chunk[["ciclo_id", "bloque_base", "fecha_sp"]].drop_duplicates().copy()
    cy["bloque_base"] = _canon_str(cy["bloque_base"])
    cy["fecha_sp"] = _to_date(cy["fecha_sp"])

    m = cy.merge(cl, on="bloque_base", how="left")
    m = m.loc[(m["fecha"] >= m["fecha_sp"]) & (m["fecha"] <= as_of_date), :].copy()

    # IMPORTANT: usar gdc_dia (acumular). Ignorar gdc_base.
    m["gdc_dia"] = pd.to_numeric(m["gdc_dia"], errors="coerce").fillna(0.0)
    m["rainfall_mm_dia"] = pd.to_numeric(m["rainfall_mm_dia"], errors="coerce").fillna(0.0)
    m["solar_energy_j_m2_dia"] = pd.to_numeric(m["solar_energy_j_m2_dia"], errors="coerce").fillna(0.0)
    m["temp_avg_dia"] = pd.to_numeric(m["temp_avg_dia"], errors="coerce")
    m["en_lluvia_dia"] = pd.to_numeric(m["en_lluvia_dia"], errors="coerce").fillna(0.0)

    m = m.sort_values(["ciclo_id", "fecha"])

    def _roll_sum(s: pd.Series, w: int) -> pd.Series:
        return s.rolling(window=w, min_periods=1).sum()
    def _roll_mean(s: pd.Series, w: int) -> pd.Series:
        return s.rolling(window=w, min_periods=1).mean()

    m["gdc_7d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 7))
    m["gdc_14d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 14))
    m["rain_7d"] = m.groupby("ciclo_id")["rainfall_mm_dia"].transform(lambda s: _roll_sum(s, 7))
    m["solar_7d"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].transform(lambda s: _roll_sum(s, 7))
    m["enlluvia_days_7d"] = m.groupby("ciclo_id")["en_lluvia_dia"].transform(lambda s: _roll_sum(s, 7))
    m["temp_avg_7d"] = m.groupby("ciclo_id")["temp_avg_dia"].transform(lambda s: _roll_mean(s, 7))

    m["gdc_cum_sp"] = m.groupby("ciclo_id")["gdc_dia"].cumsum()
    m["rain_cum_sp"] = m.groupby("ciclo_id")["rainfall_mm_dia"].cumsum()
    m["solar_cum_sp"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].cumsum()

    last = m.groupby("ciclo_id", as_index=False).tail(1).copy()
    last["days_from_sp"] = (as_of_date - last["fecha_sp"]).dt.days
    last["days_from_sp"] = pd.to_numeric(last["days_from_sp"], errors="coerce").fillna(0).clip(lower=0)
    last["gdc_per_day"] = _safe_div(last["gdc_cum_sp"], last["days_from_sp"].replace(0, np.nan))

    feat_cols = [
        "ciclo_id",
        "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
        "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
        "solar_cum_sp", "solar_7d",
        "temp_avg_7d",
    ]
    return last[feat_cols]


def main() -> None:
    ciclo = read_parquet(IN_CICLO).copy()
    grid = read_parquet(IN_GRID_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    ciclo["bloque_base"] = _canon_str(ciclo["bloque_base"])
    ciclo["fecha_sp"] = _to_date(ciclo["fecha_sp"])
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])
    ciclo["fecha_fin_cosecha"] = _to_date(ciclo["fecha_fin_cosecha"])

    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

    head = _build_cycle_header(grid)
    df0 = ciclo.merge(head, on="ciclo_id", how="inner", suffixes=("", "_ml1"))

    df0 = df0.loc[df0["fecha_inicio_cosecha"].notna() & df0["harvest_start_pred"].notna(), :].copy()

    # Panel as_of
    rows = []
    for r in df0.itertuples(index=False):
        for a in _make_asof_dates(r.fecha_sp, r.fecha_inicio_cosecha):
            rows.append((r.ciclo_id, a))
    asof_df = pd.DataFrame(rows, columns=["ciclo_id", "as_of_date"])

    df = df0.merge(asof_df, on="ciclo_id", how="inner")
    df = df.loc[df["as_of_date"] < df["fecha_inicio_cosecha"], :].copy()

    df["error_start_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_pred"]).dt.days.astype(int)

    df["dow"] = df["as_of_date"].dt.dayofweek
    df["month"] = df["as_of_date"].dt.month
    df["weekofyear"] = df["as_of_date"].dt.isocalendar().week.astype(int)

    feats = []
    for as_of_date, chunk in df.groupby("as_of_date"):
        feats.append(_features_asof(clima, chunk[["ciclo_id", "bloque_base", "fecha_sp"]], as_of_date))
    feat = pd.concat(feats, ignore_index=True) if feats else pd.DataFrame(columns=["ciclo_id"])

    df = df.merge(feat, on="ciclo_id", how="left")

    for c in [
        "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
        "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
        "solar_cum_sp", "solar_7d",
    ]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

    if "temp_avg_7d" in df.columns:
        df["temp_avg_7d"] = pd.to_numeric(df["temp_avg_7d"], errors="coerce")
        df["temp_avg_7d"] = df["temp_avg_7d"].fillna(df["temp_avg_7d"].median())

    df["created_at"] = pd.Timestamp(datetime.now()).normalize()

    OUT_DS.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(df, OUT_DS)
    print(f"[OK] Wrote {OUT_DS} rows={len(df):,} cycles={df['ciclo_id'].nunique():,}")


if __name__ == "__main__":
    main()

----------------------------------------------------------
[30/65] FILE: \src\gold\build_ds_hidr_poscosecha_ml2_v1.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

# ? base con hidr ML1 y demás columnas
IN_BASE = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"
# ? dh_final con dh_dias_final + fecha_post_pred_final
IN_DH_FINAL = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino_final.parquet"

IN_REAL_HIDR = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_DS = GOLD / "ml2_datasets" / "ds_hidr_poscosecha_ml2_v1.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _as_of_date_today_minus_1() -> pd.Timestamp:
    return (pd.Timestamp.now().normalize() - pd.Timedelta(days=1)).normalize()


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise KeyError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")
```

```
def _factor_from_hidr_pct(hidr_pct: pd.Series) -> pd.Series:
    x = pd.to_numeric(hidr_pct, errors="coerce")
    return np.where(x.isna(), np.nan, np.where(x > 3.5, 1.0 + x / 100.0, x)).astype(float)


def main() -> None:
    as_of_date = _as_of_date_today_minus_1()
    created_at = pd.Timestamp.utcnow()

    base = read_parquet(IN_BASE).copy()
    base.columns = [str(c).strip() for c in base.columns]

    _require(base, ["fecha", "bloque_base", "variedad_canon", "grado", "destino"], "pred_poscosecha_ml2_full")
    # hidr ML1 debe existir aquí
    if "factor_hidr_ml1" not in base.columns:
        raise KeyError(f"pred_poscosecha_ml2_full no tiene factor_hidr_ml1. Columnas={list(base.columns)}")

    base["fecha"] = _to_date(base["fecha"])
    base["destino"] = _canon_str(base["destino"])
    base["grado"] = _canon_int(base["grado"])
    base["bloque_base"] = _canon_str(base["bloque_base"])
    base["variedad_canon"] = _canon_str(base["variedad_canon"])

    # filtro hoy-1
    base = base.loc[base["fecha"].notna() & (base["fecha"] <= as_of_date)].copy()

    # traer DH final
    dh = read_parquet(IN_DH_FINAL).copy()
    dh.columns = [str(c).strip() for c in dh.columns]
    _require(dh, ["fecha", "bloque_base", "variedad_canon", "grado", "destino", "dh_dias_final", "fecha_post_p
red_final"], "pred_poscosecha_ml2_dh_final")

    dh["fecha"] = _to_date(dh["fecha"])
    dh["destino"] = _canon_str(dh["destino"])
    dh["grado"] = _canon_int(dh["grado"])
    dh["bloque_base"] = _canon_str(dh["bloque_base"])
    dh["variedad_canon"] = _canon_str(dh["variedad_canon"])
    dh["fecha_post_pred_final"] = _to_date(dh["fecha_post_pred_final"])

    key = ["fecha", "bloque_base", "variedad_canon", "grado", "destino"]
    dh_take = dh[key + ["dh_dias_final", "fecha_post_pred_final"]].copy()

    uni = base.merge(dh_take, on=key, how="left")

    # features calendario por fecha_post_pred_final
    uni["dow"] = uni["fecha_post_pred_final"].dt.dayofweek.astype("Int64")
    uni["month"] = uni["fecha_post_pred_final"].dt.month.astype("Int64")
    uni["weekofyear"] = uni["fecha_post_pred_final"].dt.isocalendar().week.astype("Int64")

    # pesos por tallos si existe
    tallos_col = None
    for c in ["tallos", "tallos_final_grado_dia", "tallos_pred_ml2_grado_dia", "tallos_pred_ml1_grado_dia"]:
        if c in uni.columns:
            tallos_col = c
            break
    uni["tallos_w"] = pd.to_numeric(uni[tallos_col], errors="coerce").fillna(0.0).clip(lower=0.0) if tallos_co
l else 1.0

    uni["factor_hidr_ml1"] = pd.to_numeric(uni["factor_hidr_ml1"], errors="coerce")

    # ---------------- REAL HIDR
    real = read_parquet(IN_REAL_HIDR).copy()
    real.columns = [str(c).strip() for c in real.columns]
    _require(real, ["fecha_post", "grado", "destino"], "fact_hidratacion_real_post_grado_destino")

    if "hidr_pct" in real.columns:
        real["factor_hidr_real"] = _factor_from_hidr_pct(real["hidr_pct"])
    else:
        _require(real, ["peso_base_g", "peso_post_g"], "fact_hidratacion_real_post_grado_destino")
        pb = pd.to_numeric(real["peso_base_g"], errors="coerce")
        pp = pd.to_numeric(real["peso_post_g"], errors="coerce")
        real["factor_hidr_real"] = np.where(pb > 0, pp / pb, np.nan)

    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    real["grado"] = _canon_int(real["grado"])

    # agregación real por llave
    if "tallos" in real.columns:
        real["tallos"] = pd.to_numeric(real["tallos"], errors="coerce").fillna(0.0)
        g = real.groupby(["fecha_post", "grado", "destino"], dropna=False)
        real2 = g.apply(
            lambda x: pd.Series({
                "factor_hidr_real": float(np.nansum(x["factor_hidr_real"] * x["tallos"]) / np.nansum(x["tallos
"])) if np.nansum(x["tallos"]) > 0 else float(np.nanmedian(x["factor_hidr_real"])),
                "tallos_real_sum": float(np.nansum(x["tallos"])),
            })
        ).reset_index()
    else:
        real2 = (
            real.groupby(["fecha_post", "grado", "destino"], dropna=False, as_index=False)
                .agg(factor_hidr_real=("factor_hidr_real", "median"))
        )
        real2["tallos_real_sum"] = np.nan

    ds = uni.merge(
        real2,
        left_on=["fecha_post_pred_final", "grado", "destino"],
        right_on=["fecha_post", "grado", "destino"],
        how="left",
    ).drop(columns=["fecha_post"], errors="ignore")

    # target log error
    eps = 1e-9
    ds["ratio_hidr"] = ds["factor_hidr_real"] / ds["factor_hidr_ml1"].clip(lower=eps)
    ds["log_error_hidr"] = np.log(ds["ratio_hidr"].clip(lower=eps))

    clip = 1.2
    ds["log_error_hidr_clipped"] = ds["log_error_hidr"].clip(lower=-clip, upper=clip)
    ds["is_in_real"] = ds["factor_hidr_real"].notna()

    ds["as_of_date"] = as_of_date
    ds["created_at"] = created_at

    OUT_DS.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(ds, OUT_DS)

    print(f"[OK] Wrote dataset: {OUT_DS}")
    print(f"     rows={len(ds):,} rows_with_real={int(ds['is_in_real'].sum()):,} as_of_date={as_of_date.date()
} clip=±{clip}")


if __name__ == "__main__":
    main()

-------------------------------------------------------
[31/65] FILE: \src\gold\build_ds_tallos_curve_ml2_v2.py
-------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]
```

```
ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"

IN_GRID_ML2 = GOLD_DIR / "universe_harvest_grid_ml2.parquet"
IN_REAL = SILVER_DIR / "fact_cosecha_real_grado_dia.parquet"
IN_PRED_ML1 = GOLD_DIR / "pred_tallos_grado_dia_ml1_full.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"

OUT_DS = GOLD_DIR / "ml2_datasets" / "ds_tallos_curve_ml2_v2.parquet"

EPS = 1.0
CLIP_LOG_ERR = 1.5  # exp(±1.5) ? factor [0.22 .. 4.48]


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def main() -> None:
    grid = read_parquet(IN_GRID_ML2).copy()
    real = read_parquet(IN_REAL).copy()
    pred = read_parquet(IN_PRED_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    # --- GRID ML2 ---
    grid["fecha"] = _to_date(grid["fecha"])
    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])
    if "estado" in grid.columns:
        grid["estado"] = _canon_str(grid["estado"])

    base_cols = [
        "ciclo_id", "fecha", "bloque_base", "variedad_canon", "area", "tipo_sp", "estado",
        "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final",
        "harvest_start_final", "harvest_end_final",
        "ml1_version",
    ]
    base_cols = [c for c in base_cols if c in grid.columns]
    grid0 = grid[base_cols].drop_duplicates(["ciclo_id", "fecha"]).copy()

    # --- REAL tallos (sum grades) ---
    real["fecha"] = _to_date(real["fecha"])

    # Resolver bloque_base desde columnas posibles
    if "bloque_base" in real.columns:
        real["bloque_base"] = _canon_str(real["bloque_base"])
    elif "bloque_padre" in real.columns:
        real["bloque_base"] = _canon_str(real["bloque_padre"])
    elif "bloque" in real.columns:
        real["bloque_base"] = _canon_str(real["bloque"])
    else:
        raise KeyError(
            "No encuentro columna de bloque en fact_cosecha_real_grado_dia. "
            "Espero una de: bloque_base / bloque_padre / bloque"
        )

    if "variedad" in real.columns:
        real["variedad"] = _canon_str(real["variedad"])

    # real: (fecha, bloque_base, variedad, grado, tallos_real)
    real_agg = (
        real.groupby(["fecha", "bloque_base"], as_index=False)
            .agg(tallos_real_dia=("tallos_real", "sum"))
    )

    # --- PRED ML1 tallos (sum grades) ---
```

```
pred["fecha"] = _to_date(pred["fecha"])
pred["bloque_base"] = _canon_str(pred["bloque_base"])
if "variedad_canon" in pred.columns:
    pred["variedad_canon"] = _canon_str(pred["variedad_canon"])

# pred expected: (fecha, bloque_base, variedad_canon, grado, tallos_pred_*)
# Detect the predicted column (fallback order)
# En este proyecto, la columna canónica ML1 a nivel grado-día es:
# tallos_pred_ml1_grado_dia (sumar por grado -> total día)
if "tallos_pred_ml1_grado_dia" in pred.columns:
    pred_col = "tallos_pred_ml1_grado_dia"
elif "tallos_pred_ml1_dia" in pred.columns:
    # fallback (menos preferido)
    pred_col = "tallos_pred_ml1_dia"
else:
    raise KeyError(
        "No encuentro columna de tallos predicha en pred_tallos_grado_dia_ml1_full.parquet. "
        "Espero: tallos_pred_ml1_grado_dia (preferida) o tallos_pred_ml1_dia (fallback)."
    )


pred_agg = (
    pred.groupby(["fecha", "bloque_base", "variedad_canon"], as_index=False)
        .agg(tallos_pred_ml1_dia=(pred_col, "sum"))
)

# --- CLIMA diario (ya por bloque_base + fecha) ---
clima["fecha"] = _to_date(clima["fecha"])
clima["bloque_base"] = _canon_str(clima["bloque_base"])

clima_cols = [
    "fecha", "bloque_base",
    "gdc_dia", "gdc_base",
    "rainfall_mm_dia", "horas_lluvia", "en_lluvia_dia",
    "temp_avg_dia", "solar_energy_j_m2_dia",
    "wind_speed_avg_dia", "wind_run_dia",
]
clima_cols = [c for c in clima_cols if c in clima.columns]
clima0 = clima[clima_cols].copy()


# --- JOIN ---
df = (
    grid0
    .merge(pred_agg, on=["fecha", "bloque_base", "variedad_canon"], how="left")
    .merge(real_agg, on=["fecha", "bloque_base"], how="left")
    .merge(clima0, on=["fecha", "bloque_base"], how="left")
)

# pyarrow no permite columnas duplicadas (ej: gdc_base repetida)
df = df.loc[:, ~df.columns.duplicated()].copy()


df["tallos_pred_ml1_dia"] = pd.to_numeric(df["tallos_pred_ml1_dia"], errors="coerce").fillna(0.0)
df["tallos_real_dia"] = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0)

# Target: log error ratio
ratio = (df["tallos_real_dia"] + EPS) / (df["tallos_pred_ml1_dia"] + EPS)
df["log_error"] = np.log(ratio).clip(-CLIP_LOG_ERR, CLIP_LOG_ERR)

# convenience columns
df["error_ratio"] = np.exp(df["log_error"])

# Filter: para entrenamiento, necesitamos pred > 0 o real > 0 (para no meter basura total)
df = df.loc[(df["tallos_pred_ml1_dia"] > 0) | (df["tallos_real_dia"] > 0), :].copy()

df["created_at"] = pd.Timestamp(datetime.now()).normalize()

OUT_DS.parent.mkdir(parents=True, exist_ok=True)
write_parquet(df, OUT_DS)

print(f"[OK] Wrote dataset: {OUT_DS}")
```

```
    print(f"     rows={len(df):,} cycles={df['ciclo_id'].nunique():,} "
          f"fecha_range=[{df['fecha'].min().date()}..{df['fecha'].max().date()}]")
    print(f"     pred_col_used={pred_col}")


if __name__ == "__main__":
    main()

--------------------------------------------------
[32/65] FILE: \src\gold\build_pred_kg_cajas_ml2.py
--------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"

# ---- INPUTS ML2 (backtest) ----
IN_TALLOS_ML2_BT = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"
IN_PESO_ML2_BT = EVAL / "backtest_pred_peso_tallo_grado_dia_ml2_final.parquet"

# ---- INPUTS ML2 (prod - si luego los materializas en gold) ----
IN_TALLOS_ML2_PROD = GOLD / "pred_tallos_grado_dia_ml2_full.parquet"          # opcional futuro
IN_PESO_ML2_PROD = GOLD / "pred_peso_tallo_grado_dia_ml2_full.parquet"       # opcional futuro

# ---- INPUTS ML1 (para ?heredar? conversión a cajas) ----
IN_KG_ML1 = GOLD / "pred_kg_grado_dia_ml1_full.parquet"
IN_CAJAS_ML1 = GOLD / "pred_cajas_grado_dia_ml1_full.parquet"

# ---- OUTPUTS ML2 ----
OUT_KG_GRADO = GOLD / "pred_kg_grado_dia_ml2_full.parquet"
OUT_KG_DIA = GOLD / "pred_kg_dia_ml2_full.parquet"
OUT_CAJAS_GRADO = GOLD / "pred_cajas_grado_dia_ml2_full.parquet"
OUT_CAJAS_DIA = GOLD / "pred_cajas_dia_ml2_full.parquet"


KEYS = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _pick_col(df: pd.DataFrame, candidates: list[str]) -> str:
    for c in candidates:
        if c in df.columns:
            return c
    raise KeyError(f"No encuentro ninguna de estas columnas: {candidates}. Tengo: {list(df.columns)}")


def main(mode: str = "backtest") -> None:
    mode = (mode or "backtest").strip().lower()
    if mode not in ("backtest", "prod"):
        raise ValueError("--mode debe ser backtest o prod")
    # ---- Load ML2 ----
    if mode == "backtest":
        tallos = read_parquet(IN_TALLOS_ML2_BT).copy()
        peso = read_parquet(IN_PESO_ML2_BT).copy()
    else:
        tallos = read_parquet(IN_TALLOS_ML2_PROD).copy()
        peso = read_parquet(IN_PESO_ML2_PROD).copy()

    # Canon keys
    for df in (tallos, peso):
        df["fecha"] = _to_date(df["fecha"])
        df["bloque_base"] = _canon_str(df["bloque_base"])
        df["variedad_canon"] = _canon_str(df["variedad_canon"])
        df["grado"] = _canon_str(df["grado"])

    # ---- resolve ML2 columns ----
    col_tallos_ml2 = _pick_col(
        tallos,
        ["tallos_final_grado_dia", "tallos_ml2_grado_dia", "tallos_pred_ml2_grado_dia"]
    )
    # peso backtest normalmente trae algo tipo peso_tallo_final_g
    col_peso_ml2 = _pick_col(
        peso,
        ["peso_tallo_final_g", "peso_tallo_ml2_g", "peso_tallo_pred_ml2_g"]
    )

    tallos_use = tallos[KEYS + [col_tallos_ml2]].copy().rename(columns={col_tallos_ml2: "tallos_ml2_grado_dia"
})
    peso_use = peso[KEYS + [col_peso_ml2]].copy().rename(columns={col_peso_ml2: "peso_tallo_ml2_g"})

    # ---- Merge ML2 tallos + peso ----
    df = tallos_use.merge(peso_use, on=KEYS, how="left")

    df["tallos_ml2_grado_dia"] = pd.to_numeric(df["tallos_ml2_grado_dia"], errors="coerce").fillna(0.0)
    df["peso_tallo_ml2_g"] = pd.to_numeric(df["peso_tallo_ml2_g"], errors="coerce")

    # KG ML2
    df["kg_ml2_grado_dia"] = df["tallos_ml2_grado_dia"] * df["peso_tallo_ml2_g"] / 1000.0
    df.loc[df["kg_ml2_grado_dia"] < 0, "kg_ml2_grado_dia"] = 0.0

    # ---- Load ML1 kg/cajas for conversion ----
    kg1 = read_parquet(IN_KG_ML1).copy()
    cajas1 = read_parquet(IN_CAJAS_ML1).copy()

    for df1 in (kg1, cajas1):
        df1["fecha"] = _to_date(df1["fecha"])
        df1["bloque_base"] = _canon_str(df1["bloque_base"])
        df1["variedad_canon"] = _canon_str(df1["variedad_canon"])
        df1["grado"] = _canon_str(df1["grado"])

    col_kg_ml1 = _pick_col(
        kg1,
        ["kg_pred_ml1_grado_dia", "kg_ml1_grado_dia", "kg_pred_grado_dia", "kg_pred_ml1"]
    )
    col_cajas_ml1 = _pick_col(
        cajas1,
        ["cajas_pred_ml1_grado_dia", "cajas_ml1_grado_dia", "cajas_pred_grado_dia", "cajas_pred_ml1"]
    )

    kg1_use = kg1[KEYS + [col_kg_ml1]].copy().rename(columns={col_kg_ml1: "kg_ml1_grado_dia"})
    cajas1_use = cajas1[KEYS + [col_cajas_ml1]].copy().rename(columns={col_cajas_ml1: "cajas_ml1_grado_dia"})

    df = df.merge(kg1_use, on=KEYS, how="left").merge(cajas1_use, on=KEYS, how="left")

    df["kg_ml1_grado_dia"] = pd.to_numeric(df["kg_ml1_grado_dia"], errors="coerce").fillna(0.0)
    df["cajas_ml1_grado_dia"] = pd.to_numeric(df["cajas_ml1_grado_dia"], errors="coerce").fillna(0.0)

    # Cajas ML2 heredando conversión ML1
    eps = 1e-9
    ratio = np.where(df["kg_ml1_grado_dia"] > 0, df["kg_ml2_grado_dia"] / (df["kg_ml1_grado_dia"] + eps), 0.0)
    df["cajas_ml2_grado_dia"] = df["cajas_ml1_grado_dia"] * ratio
    df.loc[df["cajas_ml2_grado_dia"] < 0, "cajas_ml2_grado_dia"] = 0.0
    # ---- Build outputs (grado/día) ----
    out_kg_grado = df[KEYS + ["kg_ml2_grado_dia"]].copy()
    out_cajas_grado = df[KEYS + ["cajas_ml2_grado_dia"]].copy()

    created_at = pd.Timestamp(datetime.now()).normalize()
    out_kg_grado["created_at"] = created_at
    out_cajas_grado["created_at"] = created_at

    # ---- Aggregate to día (sum grados) ----
    keys_day = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    out_kg_day = out_kg_grado.groupby(keys_day, as_index=False).agg(kg_ml2_dia=("kg_ml2_grado_dia", "sum"))
    out_cajas_day = out_cajas_grado.groupby(keys_day, as_index=False).agg(cajas_ml2_dia=("cajas_ml2_grado_dia"
, "sum"))
    out_kg_day["created_at"] = created_at
    out_cajas_day["created_at"] = created_at

    GOLD.mkdir(parents=True, exist_ok=True)
    write_parquet(out_kg_grado, OUT_KG_GRADO)
    write_parquet(out_kg_day, OUT_KG_DIA)
    write_parquet(out_cajas_grado, OUT_CAJAS_GRADO)
    write_parquet(out_cajas_day, OUT_CAJAS_DIA)

    print(f"[OK] Wrote: {OUT_KG_GRADO} rows={len(out_kg_grado)}")
    print(f"[OK] Wrote: {OUT_KG_DIA} rows={len(out_kg_day)}")
    print(f"[OK] Wrote: {OUT_CAJAS_GRADO} rows={len(out_cajas_grado)}")
    print(f"[OK] Wrote: {OUT_CAJAS_DIA} rows={len(out_cajas_day)}")
    print(f"     mode={mode}  tallos_col={col_tallos_ml2}  peso_col={col_peso_ml2}")
    print(f"     kg_ml1_col={col_kg_ml1}  cajas_ml1_col={col_cajas_ml1}")


if __name__ == "__main__":
    import argparse

    p = argparse.ArgumentParser()
    p.add_argument("--mode", default="backtest", choices=["backtest", "prod"])
    args = p.parse_args()
    main(mode=args.mode)

----------------------------------------------------------------
[33/65] FILE: \src\gold\build_pred_poscosecha_ml2_final_views.py
----------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"

IN_FULL = GOLD / "pred_poscosecha_ml2_ajuste_grado_dia_bloque_destino_final.parquet"

OUT_FULL = GOLD / "pred_poscosecha_ml2_final_full_grado_dia_bloque_destino.parquet"
OUT_DBD = GOLD / "pred_poscosecha_ml2_final_dia_bloque_destino.parquet"
OUT_DD = GOLD / "pred_poscosecha_ml2_final_dia_destino.parquet"
OUT_DT = GOLD / "pred_poscosecha_ml2_final_dia_total.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")


def main() -> None:
    df = read_parquet(IN_FULL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    _require(
        df,
        ["fecha", "destino", "cajas_split_grado_dia", "factor_hidr_final", "factor_desp_final", "factor_ajuste
_final"],
        "pred_poscosecha_ml2_ajuste...final",
    )

    df["fecha"] = _to_date(df["fecha"])
    df["cajas_split_grado_dia"] = pd.to_numeric(df["cajas_split_grado_dia"], errors="coerce").fillna(0.0)
    for c in ["factor_hidr_final", "factor_desp_final", "factor_ajuste_final"]:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(1.0)

    df["cajas_postcosecha_ml2_final"] = (
        df["cajas_split_grado_dia"].astype(float)
        * df["factor_hidr_final"].astype(float)
        * df["factor_desp_final"].astype(float)
        * df["factor_ajuste_final"].astype(float)
    )

    df["created_at"] = pd.Timestamp.utcnow()

    write_parquet(df, OUT_FULL)
    print(f"[OK] Wrote: {OUT_FULL} rows={len(df):,}")

    # día-bloque-destino
    keys_dbd = ["fecha", "bloque_base", "destino"]
    cols_av = [c for c in keys_dbd if c in df.columns]
    if len(cols_av) == 3:
        out_dbd = (
            df.groupby(keys_dbd, dropna=False, as_index=False)
              .agg(
                  cajas_split=("cajas_split_grado_dia", "sum"),
                  cajas_post_ml2=("cajas_postcosecha_ml2_final", "sum"),
              )
        )
        out_dbd["created_at"] = pd.Timestamp.utcnow()
        write_parquet(out_dbd, OUT_DBD)
        print(f"[OK] Wrote: {OUT_DBD} rows={len(out_dbd):,}")
    else:
        print("[WARN] No se pudo escribir OUT_DBD (faltan columnas de bloque_base).")

    # día-destino
    out_dd = (
        df.groupby(["fecha", "destino"], dropna=False, as_index=False)
          .agg(
              cajas_split=("cajas_split_grado_dia", "sum"),
              cajas_post_ml2=("cajas_postcosecha_ml2_final", "sum"),
          )
    )
    out_dd["created_at"] = pd.Timestamp.utcnow()
    write_parquet(out_dd, OUT_DD)
    print(f"[OK] Wrote: {OUT_DD} rows={len(out_dd):,}")

    # día-total
    out_dt = (
        out_dd.groupby(["fecha"], dropna=False, as_index=False)
              .agg(
                  cajas_split=("cajas_split", "sum"),
                  cajas_post_ml2=("cajas_post_ml2", "sum"),
              )
    )
    out_dt["created_at"] = pd.Timestamp.utcnow()
    write_parquet(out_dt, OUT_DT)
    print(f"[OK] Wrote: {OUT_DT} rows={len(out_dt):,}")


if __name__ == "__main__":
    main()

----------------------------------------------------------------------
[34/65] FILE: \src\gold\build_pred_poscosecha_ml2_seed_mix_grado_dia.py
----------------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

# INPUTS
IN_CAJAS_GRADO = GOLD / "pred_cajas_grado_dia_ml2_full.parquet"
IN_MIX = SILVER / "dim_mix_proceso_semana.parquet"

# OUTPUTS (ML2 seed)
OUT_GD_BD = GOLD / "pred_poscosecha_ml2_seed_grado_dia_bloque_destino.parquet"
OUT_DD = GOLD / "pred_poscosecha_ml2_seed_dia_destino.parquet"
OUT_DT = GOLD / "pred_poscosecha_ml2_seed_dia_total.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise KeyError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")


def _safe_float(x, default: float) -> float:
    try:
        v = float(x)
        if np.isfinite(v):
            return v
    except Exception:
        pass
    return float(default)


def _semana_ventas_from_fecha_cosecha(fecha: pd.Series) -> pd.Series:
    """
    CONSISTENTE con tu lógica de ventas (Fecha_Clasificacion = Fecha - 2; semana_ventas(x): (x+2)->%U).
    Para fecha de cosecha (equivale a Fecha_Clasificacion):
      Semana_Ventas = semana_ventas(fecha)
    """
    d = pd.to_datetime(fecha, errors="coerce") + pd.Timedelta(days=2)
    yy = (d.dt.year.astype("Int64") % 100).astype("Int64")
    ww = d.dt.strftime("%U").astype(int)
    ww = np.where(ww == 0, 1, ww)
    return yy.astype(str).str.zfill(2) + pd.Series(ww).astype(str).str.zfill(2)


def _collapse_sum(df: pd.DataFrame, keys: list[str], val_cols: list[str], name: str) -> pd.DataFrame:
    dup = int(df.duplicated(subset=keys).sum())
    if dup > 0:
        agg = {c: "sum" for c in val_cols if c in df.columns}
        extra = [c for c in df.columns if c not in keys and c not in agg]
        for c in extra:
            agg[c] = "first"
        out = df.groupby(keys, dropna=False, as_index=False).agg(agg)
        print(f"[WARN] {name}: duplicados por {keys} -> colapsado sum -> rows={len(out):,}")
        return out
    return df


def _resolve_cajas_col(df: pd.DataFrame) -> str:
    # prioridad: columna final ML2
    candidates = [
        "cajas_ml2_grado_dia",
        "cajas_final_grado_dia",
        "cajas_pred_ml2_grado_dia",
        "cajas_grado_dia_ml2",
        # fallback: si tu builder dejó el mismo nombre ml1 (no ideal, pero posible)
        "cajas_ml1_grado_dia",
    ]
    for c in candidates:
        if c in df.columns:
            return c
    raise KeyError(
        "No encuentro columna de cajas en pred_cajas_grado_dia_ml2_full.parquet. "
        f"Probé: {candidates}. Disponibles={list(df.columns)}"
    )


def main(as_of_date: str | None = None) -> None:
    created_at = pd.Timestamp.utcnow()

    # as_of_date default = hoy-1 (operativo)
    if as_of_date is None:
        as_of = pd.Timestamp.now().normalize() - pd.Timedelta(days=1)
    else:
        as_of = pd.to_datetime(as_of_date, errors="coerce").normalize()
        if pd.isna(as_of):
            raise ValueError(f"as_of_date inválida: {as_of_date}")

    # -------------------------
    # 1) Load cajas ML2 (grado/día) -> supply
    # -------------------------
    cajas = read_parquet(IN_CAJAS_GRADO).copy()
    cajas.columns = [str(c).strip() for c in cajas.columns]

    need = ["fecha", "bloque_base", "variedad_canon", "grado"]
    _require(cajas, need, "pred_cajas_grado_dia_ml2_full")

    cajas["fecha"] = _to_date(cajas["fecha"])
    cajas["bloque_base"] = _canon_int(cajas["bloque_base"])
    cajas["grado"] = _canon_int(cajas["grado"])
    cajas["variedad_canon"] = _canon_str(cajas["variedad_canon"])

    cajas_col = _resolve_cajas_col(cajas)
    cajas["cajas_campo_ml2_grado_dia"] = pd.to_numeric(cajas[cajas_col], errors="coerce").fillna(0.0)

    # filtro operativo hoy-1
    cajas = cajas.loc[cajas["fecha"].notna() & (cajas["fecha"] <= as_of)].copy()

    key_supply = ["fecha", "bloque_base", "variedad_canon", "grado"]
    cajas = _collapse_sum(cajas[key_supply + ["cajas_campo_ml2_grado_dia"]], key_supply, ["cajas_campo_ml2_gra
do_dia"], "cajas_ml2")

    # -------------------------
    # 2) Load mix por Semana_Ventas
    # -------------------------
    mix = read_parquet(IN_MIX).copy()
    mix.columns = [str(c).strip() for c in mix.columns]
    _require(mix, ["Semana_Ventas", "W_Blanco", "W_Arcoiris", "W_Tinturado"], "dim_mix_proceso_semana")

    mix["Semana_Ventas"] = mix["Semana_Ventas"].astype(str).str.strip()
    for c in ["W_Blanco", "W_Arcoiris", "W_Tinturado"]:
        mix[c] = pd.to_numeric(mix[c], errors="coerce")

    # DEFAULT obligatorio
    mix_def = mix[mix["Semana_Ventas"].eq("DEFAULT")].copy()
    if mix_def.empty:
        raise ValueError("dim_mix_proceso_semana no trae Semana_Ventas='DEFAULT' (fallback obligatorio).")
    r = mix_def.iloc[0]
    def_w = {
        "W_Blanco": _safe_float(r["W_Blanco"], 0.80),
        "W_Arcoiris": _safe_float(r["W_Arcoiris"], 0.10),
        "W_Tinturado": _safe_float(r["W_Tinturado"], 0.10),
    }
    sdef = def_w["W_Blanco"] + def_w["W_Arcoiris"] + def_w["W_Tinturado"]
    def_w = {k: (v / sdef if sdef > 0 else v) for k, v in def_w.items()}

    DESTS = [
        ("BLANCO", "W_Blanco"),
        ("ARCOIRIS", "W_Arcoiris"),
        ("TINTURADO", "W_Tinturado"),
    ]

    # -------------------------
    # 3) Semana_Ventas + join mix (fallback DEFAULT)
    # -------------------------
    supply = cajas.copy()
    supply["Semana_Ventas"] = _semana_ventas_from_fecha_cosecha(supply["fecha"])

    mix_take = mix[mix["Semana_Ventas"].ne("DEFAULT")].drop_duplicates(subset=["Semana_Ventas"], keep="last").
copy()
    supply = supply.merge(mix_take, on="Semana_Ventas", how="left")

    for _, wcol in DESTS:
        supply[wcol] = supply[wcol].fillna(def_w[wcol])

    # renormalize defensivo
    ws = supply[[w for _, w in DESTS]].sum(axis=1).replace(0, np.nan)
    for _, wcol in DESTS:
        supply[wcol] = np.where(ws.notna(), supply[wcol] / ws, def_w[wcol])

    # -------------------------
    # 4) Split por destino (mass-balance exact)
    # -------------------------
    chunks = []
    for dest, wcol in DESTS:
        sub = supply[key_supply + ["Semana_Ventas", "cajas_campo_ml2_grado_dia", wcol]].copy()
        sub = sub.rename(columns={wcol: "w_dest"})
        sub["destino"] = dest
        sub["cajas_split_grado_dia"] = sub["cajas_campo_ml2_grado_dia"].astype(float) * sub["w_dest"].astype(f
loat)
        chunks.append(sub)

    split = pd.concat(chunks, ignore_index=True)
    split["destino"] = _canon_str(split["destino"])

    chk = (
        split.groupby(key_supply, dropna=False, as_index=False)
            .agg(sum_split=("cajas_split_grado_dia", "sum"))
            .merge(supply[key_supply + ["cajas_campo_ml2_grado_dia"]], on=key_supply, how="left", validate="1:
1")
    )
    chk["abs_diff"] = (chk["sum_split"] - chk["cajas_campo_ml2_grado_dia"]).abs()
    max_abs = float(chk["abs_diff"].max()) if len(chk) else 0.0
    print(f"[CHECK] split mass-balance max_abs_diff={max_abs:.12f}")
    if max_abs > 1e-9:
        raise ValueError("[FATAL] split por destino no conserva cajas (mass-balance roto).")

    split["as_of_date"] = as_of
    split["cajas_base_col_used"] = cajas_col
    split["created_at"] = created_at

    # -------------------------
    # Outputs
    # -------------------------
    out_gd_bd = split[
        [
            "fecha",
            "bloque_base",
            "variedad_canon",
            "grado",
            "Semana_Ventas",
            "destino",
            "cajas_campo_ml2_grado_dia",
            "cajas_split_grado_dia",
            "as_of_date",
            "cajas_base_col_used",
            "created_at",
        ]
    ].copy()

    GOLD.mkdir(parents=True, exist_ok=True)
    write_parquet(out_gd_bd, OUT_GD_BD)
    print(f"[OK] Wrote: {OUT_GD_BD} rows={len(out_gd_bd):,} as_of_date={as_of.date()}")

    out_dd = (
        out_gd_bd.groupby(["fecha", "destino"], dropna=False, as_index=False)
            .agg(
                cajas_campo_ml2_dia=("cajas_campo_ml2_grado_dia", "sum"),
                cajas_split_dia=("cajas_split_grado_dia", "sum"),
            )
    )
    out_dd["as_of_date"] = as_of
    out_dd["created_at"] = created_at
    write_parquet(out_dd, OUT_DD)
    print(f"[OK] Wrote: {OUT_DD} rows={len(out_dd):,}")

    out_dt = (
        out_dd.groupby(["fecha"], dropna=False, as_index=False)
            .agg(
                cajas_campo_ml2_dia=("cajas_campo_ml2_dia", "sum"),
                cajas_split_dia=("cajas_split_dia", "sum"),
            )
    )
    out_dt["as_of_date"] = as_of
    out_dt["created_at"] = created_at
    write_parquet(out_dt, OUT_DT)
    print(f"[OK] Wrote: {OUT_DT} rows={len(out_dt):,}")


if __name__ == "__main__":
    main()

-----------------------------------------------------
[35/65] FILE: \src\gold\build_pred_tallos_ml2_full.py
-----------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd

from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"

# Backtest source (lo que ya tienes)
IN_TALLOS_ML2_BT = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"

# Output GOLD (vista final)
OUT_TALLOS_GRADO = GOLD / "pred_tallos_grado_dia_ml2_full.parquet"
OUT_TALLOS_DIA = GOLD / "pred_tallos_dia_ml2_full.parquet"


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def main() -> None:
    df = read_parquet(IN_TALLOS_ML2_BT).copy()

    # Canon
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_str(df["bloque_base"])
    df["variedad_canon"] = _canon_str(df["variedad_canon"])
    df["grado"] = _canon_str(df["grado"])

    # Resolver columna de tallos final
    if "tallos_final_grado_dia" not in df.columns:
        raise KeyError(f"No encuentro tallos_final_grado_dia en {IN_TALLOS_ML2_BT}. Columnas: {list(df.columns
)}")

    out = df[["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "tallos_final_grado_dia"]].copy()
    out = out.rename(columns={"tallos_final_grado_dia": "tallos_pred_ml2_grado_dia"})
    out["created_at"] = pd.Timestamp(datetime.now()).normalize()

    # Dia total
    keys_day = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    out_day = out.groupby(keys_day, as_index=False).agg(tallos_pred_ml2_dia=("tallos_pred_ml2_grado_dia", "sum
"))
    out_day["created_at"] = out["created_at"].iloc[0]

    GOLD.mkdir(parents=True, exist_ok=True)
    write_parquet(out, OUT_TALLOS_GRADO)
    write_parquet(out_day, OUT_TALLOS_DIA)

    print(f"[OK] Wrote: {OUT_TALLOS_GRADO} rows={len(out)}")
    print(f"[OK] Wrote: {OUT_TALLOS_DIA} rows={len(out_day)}")


if __name__ == "__main__":
    main()

------------------------------------------------------------
[36/65] FILE: \src\gold\postprocess_curva_share_smooth_ml1.py
------------------------------------------------------------
from __future__ import annotations

from pathlib import Path

import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet
```

```
# =============================================================================
# Paths
# =============================================================================
PRED_FACTOR_PATH = Path("data/gold/pred_factor_curva_ml1.parquet")
FEATURES_CURVA_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")

OUT_PATH = Path("data/gold/pred_factor_curva_ml1.parquet")  # overwrite downstream-safe

# =============================================================================
# Hyperparams (estadísticos + smoothing)
# =============================================================================
# cap por segmento = quantil alto del share_real (por defecto P99.5)
CAP_Q = 0.995

# buckets de duración (días) para estabilizar caps
NDAYS_BINS = [0, 25, 35, 45, 55, 65, 80, 120, 10_000]
NDAYS_LABELS = ["<=25", "26-35", "36-45", "46-55", "56-65", "66-80", "81-120", "120+"]

# smoothing (share-space)
SMOOTH_WIN = 5          # 3 o 5 recomendado
SMOOTH_CENTER = True    # centered rolling
SMOOTH_MINP = 1

# factor safety (mantener compat)
FACTOR_MIN, FACTOR_MAX = 0.2, 5.0
EPS = 1e-9


# =============================================================================
# Helpers
# =============================================================================
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int64(s: pd.Series) -> pd.Series:
    # usamos Int64 nullable para evitar merges raros con NaNs
    x = pd.to_numeric(s, errors="coerce")
    return x.astype("Int64")


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Cols={list(df.columns)}")


def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))


def _ndays_bucket(n: pd.Series) -> pd.Series:
    n2 = pd.to_numeric(n, errors="coerce")
    return pd.cut(n2, bins=NDAYS_BINS, labels=NDAYS_LABELS, right=True, include_lowest=True).astype(str)


def _rolling_smooth_share(df: pd.DataFrame, share_col: str, out_col: str) -> pd.DataFrame:
    """
    Suaviza share por ciclo en el orden temporal de fecha.
    """
    out = df.copy()
    out = out.sort_values(["ciclo_id", "fecha"], kind="mergesort")
    out[out_col] = (
        out.groupby("ciclo_id", dropna=False)[share_col]
        .transform(lambda s: s.rolling(SMOOTH_WIN, center=SMOOTH_CENTER, min_periods=SMOOTH_MINP).mean())
    )
    return out


# =============================================================================
# Main
# =============================================================================
def main() -> None:
    created_at = pd.Timestamp.now("UTC")

    # -------------------------
    # Read inputs
    # -------------------------
    for p in [PRED_FACTOR_PATH, FEATURES_CURVA_PATH, UNIVERSE_PATH, PROG_PATH, DIM_VAR_PATH]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    pred = read_parquet(PRED_FACTOR_PATH).copy()
    feat = read_parquet(FEATURES_CURVA_PATH).copy()
    uni = read_parquet(UNIVERSE_PATH).copy()
    prog = read_parquet(PROG_PATH).copy()
    dim_var = read_parquet(DIM_VAR_PATH).copy()

    var_map = _load_var_map(dim_var)

    # -------------------------
    # Canon llaves
    # -------------------------
    _require(pred, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "pred_factor_curva_ml1")
    pred["ciclo_id"] = pred["ciclo_id"].astype(str)
    pred["fecha"] = _to_date(pred["fecha"])
    pred["bloque_base"] = _canon_int64(pred["bloque_base"])
    pred["variedad_canon"] = _canon_str(pred["variedad_canon"])

    _require(
        feat,
        ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "tallos_proy"],
        "features_curva",
    )
    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["bloque_base"] = _canon_int64(feat["bloque_base"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            feat[c] = _canon_str(feat[c])

    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_ml1")
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int64(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    _require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "dim_cosecha_progress_
bloque_fecha")
    prog["ciclo_id"] = prog["ciclo_id"].astype(str)
    prog["fecha"] = _to_date(prog["fecha"])
    prog["bloque_base"] = _canon_int64(prog["bloque_base"])
    prog["variedad_raw"] = _canon_str(prog["variedad"])
    prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])
    prog["variedad_canon"] = _canon_str(prog["variedad_canon"])
    prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]

    # -------------------------
    # Universe day-count (n_harvest_days_pred) por ciclo (para buckets)
    # -------------------------
    if "n_harvest_days_pred" in uni.columns:
        n_days = (
            uni.groupby("ciclo_id", dropna=False)["n_harvest_days_pred"]
            .max()
            .rename("n_days")
            .reset_index()
        )
    else:
        n_days = uni.groupby("ciclo_id", dropna=False)["fecha"].count().rename("n_days").reset_index()

    # -------------------------
    # Build share_real panelizado en universe (missing days = 0)
    # -------------------------
    uni_k = uni[key].drop_duplicates()
    prog_k = prog[key + ["tallos_real_dia"]].drop_duplicates(subset=key)

    real_panel = uni_k.merge(prog_k, on=key, how="left")
    real_panel["tallos_real_dia"] = pd.to_numeric(real_panel["tallos_real_dia"], errors="coerce").fillna(0.0)

    cyc_sum = real_panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
    real_panel["has_real"] = cyc_sum > 0
    real_panel["share_real"] = np.where(
        cyc_sum > 0,
        real_panel["tallos_real_dia"].astype(float) / cyc_sum,
        np.nan,
    )

    # -------------------------
    # Segment info (variedad/area/tipo_sp + bucket)
    # -------------------------
    seg_cols = ["ciclo_id", "bloque_base", "variedad_canon"]
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            seg_cols.append(c)

    seg = feat[seg_cols].drop_duplicates(subset=["ciclo_id", "bloque_base", "variedad_canon"])
    seg = seg.merge(n_days, on="ciclo_id", how="left")
    seg["ndays_bucket"] = _ndays_bucket(seg["n_days"])
    for c in ["area", "tipo_sp"]:
        if c in seg.columns:
            seg[c] = _canon_str(seg[c].fillna("UNKNOWN"))

    # -------------------------
    # Caps estadísticos (por segmento) usando share_real histórico
    # -------------------------
    real_for_caps = real_panel.merge(
        seg,
        on=["ciclo_id", "bloque_base", "variedad_canon"],
        how="left",
        suffixes=("", "_seg"),
    )

    seg_key = ["variedad_canon", "ndays_bucket"]
    if "area" in real_for_caps.columns:
        seg_key.append("area")
    if "tipo_sp" in real_for_caps.columns:
        seg_key.append("tipo_sp")

    caps_base = real_for_caps[real_for_caps["has_real"] & real_for_caps["share_real"].notna()].copy()

    # fallback global cap
    global_cap = float(caps_base["share_real"].quantile(CAP_Q)) if len(caps_base) else 1.0

    caps = (
        caps_base.groupby(seg_key, dropna=False)["share_real"]
        .quantile(CAP_Q)
        .rename("cap_share")
        .reset_index()
    )
    caps["cap_share"] = pd.to_numeric(caps["cap_share"], errors="coerce").fillna(global_cap)
    caps["cap_share"] = caps["cap_share"].clip(lower=0.0, upper=0.30)
```

```
    # -------------------------
    # Construir share_pred desde pred_factor (preferir share_curva_ml1 si existe)
    # -------------------------
    take_feat = key + ["tallos_pred_baseline_dia", "tallos_proy"]
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            take_feat.append(c)
    take_feat = list(dict.fromkeys(take_feat))

    pred2 = pred.merge(feat[take_feat].drop_duplicates(subset=key), on=key, how="left", suffixes=("", "_f"))
    pred2 = pred2.merge(n_days, on="ciclo_id", how="left")
    pred2["ndays_bucket"] = _ndays_bucket(pred2["n_days"])

    for c in ["area", "tipo_sp"]:
        if c in pred2.columns:
            pred2[c] = _canon_str(pred2[c].fillna("UNKNOWN"))

    # robust: si faltan columnas numéricas, crear con 0
    if "tallos_pred_baseline_dia" in pred2.columns:
        pred2["tallos_pred_baseline_dia"] = pd.to_numeric(pred2["tallos_pred_baseline_dia"], errors="coerce").
fillna(0.0)
    else:
        pred2["tallos_pred_baseline_dia"] = 0.0

    if "tallos_proy" in pred2.columns:
        pred2["tallos_proy"] = pd.to_numeric(pred2["tallos_proy"], errors="coerce").fillna(0.0)
    else:
        pred2["tallos_proy"] = 0.0

    if "share_curva_ml1" in pred2.columns:
        share_pred = pd.to_numeric(pred2["share_curva_ml1"], errors="coerce").fillna(0.0)
        share_pred = share_pred.clip(lower=0.0)
        pred2["share_pred_in"] = share_pred
        pred2["_share_source"] = "share_curva_ml1"
    else:
        if "factor_curva_ml1" not in pred2.columns:
            raise ValueError("pred_factor_curva_ml1 no tiene share_curva_ml1 ni factor_curva_ml1. No puedo rec
onstruir share.")
        factor = pd.to_numeric(pred2["factor_curva_ml1"], errors="coerce").fillna(1.0)
        tallos_ml1_dia = pred2["tallos_pred_baseline_dia"].astype(float) * factor.astype(float)
        denom = tallos_ml1_dia.groupby(pred2["ciclo_id"]).transform("sum").astype(float)
        pred2["share_pred_in"] = np.where(denom > 0, tallos_ml1_dia / denom, 0.0)
        pred2["_share_source"] = "factor_curva_ml1"

    # -------------------------
    # Aplicar cap estadístico + smoothing + renorm por ciclo
    # -------------------------
    # IMPORTANTÍSIMO: pred ya puede traer cap_share; si no lo botamos, el merge crea cap_share_x/cap_share_y y
 luego falla.
    if "cap_share" in pred2.columns:
        pred2 = pred2.drop(columns=["cap_share"], errors="ignore")

    pred2 = pred2.merge(caps, on=seg_key, how="left")

    # si por algo no vino cap_share tras el merge, fallback a global_cap
    if "cap_share" not in pred2.columns:
        pred2["cap_share"] = float(global_cap)
    else:
        pred2["cap_share"] = pd.to_numeric(pred2["cap_share"], errors="coerce").fillna(global_cap)

    pred2["cap_share"] = pred2["cap_share"].clip(lower=0.0, upper=0.30)

    # cap pre-smooth
    pred2["share_cap_pre"] = pd.to_numeric(pred2["share_pred_in"], errors="coerce").fillna(0.0).clip(lower=0.0
)
    pred2["was_capped_pre"] = pred2["share_cap_pre"] > pred2["cap_share"]
    pred2["share_cap_pre"] = np.minimum(pred2["share_cap_pre"], pred2["cap_share"])

    # smooth
    pred2 = _rolling_smooth_share(pred2, "share_cap_pre", "share_smooth")
    pred2["share_smooth"] = pd.to_numeric(pred2["share_smooth"], errors="coerce").fillna(0.0).clip(lower=0.0)
    # cap post-smooth
    pred2["was_capped_post"] = pred2["share_smooth"] > pred2["cap_share"]
    pred2["share_cap_post"] = np.minimum(pred2["share_smooth"], pred2["cap_share"])

    # renormalize per cycle
    s = pred2.groupby("ciclo_id", dropna=False)["share_cap_post"].transform("sum").astype(float)
    pred2["share_final"] = np.where(s > 0, pred2["share_cap_post"] / s, 0.0)

    # -------------------------
    # Reconstrucción tallos_ml1_dia y factor compatible downstream
    # -------------------------
    pred2["tallos_pred_ml1_dia_smooth"] = pred2["tallos_proy"].astype(float) * pred2["share_final"].astype(flo
at)

    base = pred2["tallos_pred_baseline_dia"].astype(float)
    pred2["factor_curva_ml1_raw_smooth"] = np.where(base > 0, pred2["tallos_pred_ml1_dia_smooth"] / (base + EP
S), 1.0)
    pred2["factor_curva_ml1_smooth"] = pred2["factor_curva_ml1_raw_smooth"].clip(lower=FACTOR_MIN, upper=FACTO
R_MAX)
    pred2["factor_curva_ml1_smooth"] = np.where(np.isfinite(pred2["factor_curva_ml1_smooth"]), pred2["factor_c
urva_ml1_smooth"], 1.0)

    # -------------------------
    # Overwrite principal factor columns (downstream)
    # -------------------------
    if "ml1_version" not in pred2.columns:
        pred2["ml1_version"] = "UNKNOWN"

    pred2["factor_curva_ml1_raw"] = pred2["factor_curva_ml1_raw_smooth"]
    pred2["factor_curva_ml1"] = pred2["factor_curva_ml1_smooth"]

    # mantener share para auditoría
    pred2["share_curva_ml1_raw"] = pred2.get("share_curva_ml1_raw", pred2["share_pred_in"])
    pred2["share_curva_ml1"] = pred2["share_final"]

    pred2["created_at"] = created_at

    # -------------------------
    # Checks
    # -------------------------
    cyc = pred2.groupby("ciclo_id", dropna=False).agg(
        proy=("tallos_proy", "max"),
        sum_ml1=("tallos_pred_ml1_dia_smooth", "sum"),
    ).reset_index()
    cyc["abs_diff"] = (cyc["proy"] - cyc["sum_ml1"]).abs()
    max_abs = float(cyc["abs_diff"].max()) if len(cyc) else float("nan")

    ss = pred2.groupby("ciclo_id", dropna=False)["share_final"].sum()
    smin, smax = (float(ss.min()), float(ss.max())) if len(ss) else (float("nan"), float("nan"))

    cap_pre = float(pred2["was_capped_pre"].mean()) if len(pred2) else float("nan")
    cap_post = float(pred2["was_capped_post"].mean()) if len(pred2) else float("nan")

    # -------------------------
    # Write output (compatible)
    # -------------------------
    keep = [
        "ciclo_id", "fecha", "bloque_base", "variedad_canon",
        "factor_curva_ml1", "factor_curva_ml1_raw", "ml1_version", "created_at",
        # auditoría (no rompe downstream)
        "_share_source", "cap_share",
        "share_pred_in", "share_smooth", "share_curva_ml1",
        "tallos_pred_ml1_dia_smooth",
        "factor_curva_ml1_raw_smooth",
        "was_capped_pre", "was_capped_post",
    ]
    keep = [c for c in keep if c in pred2.columns]

    out = pred2[keep].sort_values(["bloque_base", "variedad_canon", "fecha", "ciclo_id"]).reset_index(drop=Tru
e)
```

```
    OUT_PATH.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(out, OUT_PATH)

    print(f"OK -> {OUT_PATH} | rows={len(out):,}")
    print(f"[CHECK] mass balance vs tallos_proy | max abs diff: {max_abs:.12f}")
    print(f"[CHECK] share sum per ciclo | min={smin:.8f} max={smax:.8f}")
    print(f"[CHECK] capped rate pre={cap_pre:.4f} post={cap_post:.4f}")
    print(f"[CAP] global_cap(Q={CAP_Q}) = {global_cap:.6f}")
    print(f"[SMOOTH] win={SMOOTH_WIN} centered={SMOOTH_CENTER}")


if __name__ == "__main__":
    main()

----------------------------------------------------
[37/65] FILE: \src\models\ml1\apply_curva_cdf_dia.py
----------------------------------------------------
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet

# =============================================================================
# Paths
# =============================================================================
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/curva_cdf_dia")
OUT_PATH = Path("data/gold/pred_factor_curva_ml1.parquet")

# =============================================================================
# Model columns (must match training)
# =============================================================================
NUM_COLS = [
    "day_in_harvest",
    "rel_pos",
    "n_harvest_days",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]
CAT_COLS_MERGE = ["area", "tipo_sp"]

# =============================================================================
# Statistical adjustments (NOT capacity)
# =============================================================================
SMOOTH_WIN = 5   # rolling mean centered on share (per cycle)
EPS = 1e-12
FACTOR_MIN, FACTOR_MAX = 0.2, 5.0


# =============================================================================
# Helpers
# =============================================================================
def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)

    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base


def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df

    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)

    keep_series: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        if len(idxs) == 1:
            keep_series[c] = out.iloc[:, idxs[0]]
        else:
            s = out.iloc[:, idxs[0]]
            for j in idxs[1:]:
                s2 = out.iloc[:, j]
                s = s.where(s.notna(), s2)
            keep_series[c] = s

    ordered: list[str] = []
    for c in out.columns.astype(str):
        if c not in ordered:
            ordered.append(c)

    return pd.DataFrame({c: keep_series[c] for c in ordered})


def _baseline_share_and_cdf(tmp: pd.DataFrame, is_h_np: np.ndarray) -> tuple[np.ndarray, np.ndarray, np.ndarra
y]:
    """
    baseline_share: b_h / sum(b_h) per cycle (harvest only)
    baseline_cdf: cumsum(baseline_share) per cycle (already sorted)
    sb: sum baseline harvest per cycle (transform)
    """
    baseline = pd.to_numeric(tmp["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0).to_numpy(dtype=floa
t)
    b_h = np.where(is_h_np, baseline, 0.0)
    sb = pd.Series(b_h).groupby(tmp["ciclo_id"], dropna=False).transform("sum").to_numpy(dtype=float)
    base_share = np.where(sb > 0, b_h / sb, 0.0)

    base_cdf = pd.Series(base_share).groupby(tmp["ciclo_id"], dropna=False).cumsum().to_numpy(dtype=float)
    base_cdf = np.clip(base_cdf, 0.0, 1.0)
    return base_share, base_cdf, sb


def _smooth_share_centered(tmp: pd.DataFrame, share_col: str, is_h_col: str) -> np.ndarray:
    """
    Smooth share per cycle (harvest only) via centered rolling mean.
    Then renormalize to sum=1 per cycle (harvest only).
    """
    if SMOOTH_WIN <= 1:
        return tmp[share_col].to_numpy(dtype=float)

    df = tmp[["ciclo_id", share_col, is_h_col]].copy()
    df[share_col] = pd.to_numeric(df[share_col], errors="coerce").fillna(0.0).clip(lower=0.0)
    df.loc[~df[is_h_col], share_col] = 0.0

    sm = (
        df.groupby("ciclo_id", dropna=False)[share_col]
        .rolling(window=SMOOTH_WIN, center=True, min_periods=1)
        .mean()
        .reset_index(level=0, drop=True)
    ).to_numpy(dtype=float)

    is_h = df[is_h_col].to_numpy(dtype=bool)
    sm = np.where(is_h, sm, 0.0)

    s = pd.Series(sm).groupby(df["ciclo_id"], dropna=False).transform("sum").to_numpy(dtype=float)
    sm = np.where(s > 0, sm / s, sm)
    return sm


def _first_nonnull(s: pd.Series) -> float:
    s2 = s.dropna()
    return float(s2.iloc[0]) if len(s2) else np.nan


def _last_nonnull(s: pd.Series) -> float:
    s2 = s.dropna()
    return float(s2.iloc[-1]) if len(s2) else np.nan


# =============================================================================
# Main
# =============================================================================
def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    if not metrics_path.exists():
        raise FileNotFoundError(f"No encontré metrics.json en {ver_dir}")

    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    model_path = ver_dir / "model_curva_cdf_dia.joblib"
    if not model_path.exists():
        raise FileNotFoundError(f"No encontré modelo: {model_path}")
    model = load(model_path)

    feat = _dedupe_columns(read_parquet(FEATURES_PATH).copy())
    uni = read_parquet(UNIVERSE_PATH).copy()

    # -------------------------
    # Canon keys
    # -------------------------
    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["bloque_base"] = _canon_int(feat["bloque_base"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            feat[c] = _canon_str(feat[c])

    _coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_fina
l"])
    _coalesce_cols(feat, "rel_pos", ["rel_pos", "rel_pos_pred", "rel_pos_pred_final"])
    _coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_fina
l"])

    for c in ["day_in_harvest", "rel_pos", "n_harvest_days"]:
        feat[c] = pd.to_numeric(feat[c], errors="coerce")

    if "tallos_pred_baseline_dia" not in feat.columns:
        raise ValueError("features_curva: falta tallos_pred_baseline_dia")
    feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna
(0.0)

    if "tallos_proy" in feat.columns:
        feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0)
    else:
        feat["tallos_proy"] = 0.0

    # Ensure model cols exist
    for c in NUM_COLS:
        if c not in feat.columns:
            feat[c] = np.nan
    for c in CAT_COLS:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"

    # Universe
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key].drop_duplicates()

    feat_take = key + ["tallos_pred_baseline_dia", "tallos_proy"] + NUM_COLS + CAT_COLS_MERGE
    feat_take = list(dict.fromkeys(feat_take))
    panel = uni_k.merge(feat[feat_take], on=key, how="left")

    # Fill defaults for missing matches
    for c in NUM_COLS:
        if c not in panel.columns:
            panel[c] = np.nan
    for c in CAT_COLS:
        if c not in panel.columns:
            panel[c] = "UNKNOWN"

    panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
    if "area" in panel.columns:
        panel["area"] = _canon_str(panel["area"].fillna("UNKNOWN"))
    if "tipo_sp" in panel.columns:
        panel["tipo_sp"] = _canon_str(panel["tipo_sp"].fillna("UNKNOWN"))

    # Harvest mask
    dih = pd.to_numeric(panel["day_in_harvest"], errors="coerce")
    nh = pd.to_numeric(panel["n_harvest_days"], errors="coerce")
    is_h = dih.notna() & nh.notna() & (dih >= 1) & (nh >= 1) & (dih <= nh)
    is_h_np = is_h.to_numpy(dtype=bool)

    # One-hot aligned with training
    X = panel[NUM_COLS + CAT_COLS].copy()
    X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)
    feat_names = meta.get("feature_names", [])
    if not feat_names:
        raise ValueError("metrics.json no contiene feature_names (necesario para alinear dummies).")

    for c in feat_names:
        if c not in X.columns:
            X[c] = 0.0
    X = X[feat_names]

    # Predict CDF
    cdf_pred = model.predict(X)
    cdf_pred = pd.to_numeric(pd.Series(cdf_pred), errors="coerce").fillna(0.0).to_numpy(dtype=float)
    cdf_pred = np.clip(cdf_pred, 0.0, 1.0)
    cdf_pred = np.where(is_h_np, cdf_pred, 0.0)

    tmp = panel[key + ["tallos_pred_baseline_dia", "tallos_proy", "day_in_harvest", "rel_pos", "n_harvest_days
"]].copy()
    tmp["ml1_version"] = ver_dir.name
    tmp["cdf_pred_raw"] = cdf_pred
    tmp["is_harvest"] = is_h_np

    # Sort within cycle
    tmp["_dih_sort"] = pd.to_numeric(tmp["day_in_harvest"], errors="coerce")
    tmp["_sort_key"] = np.where(
        tmp["_dih_sort"].notna(),
        tmp["_dih_sort"].astype(float),
        tmp["fecha"].astype("int64").astype(float),
    )
    tmp = tmp.sort_values(["ciclo_id", "_sort_key"], kind="mergesort").reset_index(drop=True)

    # Monotone
    tmp["cdf_pred_mono"] = tmp.groupby("ciclo_id", dropna=False)["cdf_pred_raw"].cummax().clip(0.0, 1.0)

    # Baseline fallback (already sorted)
    base_share, base_cdf, sb = _baseline_share_and_cdf(tmp, tmp["is_harvest"].to_numpy(dtype=bool))

    # Anchor within harvest: (cdf - start)/(end-start)
    cdf_h = tmp["cdf_pred_mono"].where(tmp["is_harvest"])
    cdf_start = cdf_h.groupby(tmp["ciclo_id"], dropna=False).transform(_first_nonnull).to_numpy(dtype=float)
    cdf_end = cdf_h.groupby(tmp["ciclo_id"], dropna=False).transform(_last_nonnull).to_numpy(dtype=float)
    denom = cdf_end - cdf_start

    cdf_adj = (tmp["cdf_pred_mono"].to_numpy(dtype=float) - cdf_start) / (denom + EPS)
    cdf_adj = np.clip(cdf_adj, 0.0, 1.0)

    use_model = tmp["is_harvest"].to_numpy(dtype=bool) & np.isfinite(denom) & (denom > 1e-6)
    cdf_final = np.where(use_model, cdf_adj, base_cdf)
    cdf_final = np.where(tmp["is_harvest"].to_numpy(dtype=bool), cdf_final, 0.0)

    tmp["cdf_pred_adj"] = cdf_final
    tmp["cdf_pred_adj"] = tmp.groupby("ciclo_id", dropna=False)["cdf_pred_adj"].cummax().clip(0.0, 1.0)

    # Force end=1 inside harvest (if harvest exists)
    cdf_end2 = tmp["cdf_pred_adj"].where(tmp["is_harvest"]).groupby(tmp["ciclo_id"], dropna=False).transform(_
last_nonnull).to_numpy(dtype=float)
    scale = np.where(np.isfinite(cdf_end2) & (cdf_end2 > 1e-6), cdf_end2, 1.0)
    tmp["cdf_pred_adj"] = np.where(tmp["is_harvest"], tmp["cdf_pred_adj"] / scale, 0.0)
    tmp["cdf_pred_adj"] = tmp["cdf_pred_adj"].clip(0.0, 1.0)

    # Share = diff(CDF)
    tmp["share_pred_in"] = tmp.groupby("ciclo_id", dropna=False)["cdf_pred_adj"].diff()
    tmp["share_pred_in"] = tmp["share_pred_in"].fillna(tmp["cdf_pred_adj"]).astype(float)
    tmp["share_pred_in"] = tmp["share_pred_in"].clip(lower=0.0)
    tmp["share_pred_in"] = np.where(tmp["is_harvest"].to_numpy(dtype=bool), tmp["share_pred_in"].to_numpy(dtyp
e=float), 0.0)

    # Renormalize per cycle (harvest); if sum=0 fallback baseline share
    s = tmp.groupby("ciclo_id", dropna=False)["share_pred_in"].transform("sum").astype(float).to_numpy()
    share = np.where(s > 0, tmp["share_pred_in"].to_numpy(dtype=float) / s, base_share)
    tmp["share_curva_ml1"] = share

    # Smooth
    tmp["share_smooth"] = _smooth_share_centered(tmp, "share_curva_ml1", "is_harvest")

    # Total per cycle: prefer tallos_proy if >0 else sum baseline harvest (sb)
    tproy = pd.to_numeric(tmp["tallos_proy"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
    cyc_tproy = pd.Series(tproy).groupby(tmp["ciclo_id"], dropna=False).transform("max").to_numpy(dtype=float)
    cyc_total = np.where(cyc_tproy > 0, cyc_tproy, sb)

    tallos_ml1_dia = cyc_total * tmp["share_smooth"].to_numpy(dtype=float)
    tmp["tallos_pred_ml1_dia_from_cdf"] = tallos_ml1_dia

    baseline = pd.to_numeric(tmp["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0).to_numpy(dtype=floa
t)

    factor_raw = np.where(tmp["is_harvest"].to_numpy(dtype=bool), tallos_ml1_dia / (baseline + 1e-9), 1.0)
    factor = np.clip(factor_raw, FACTOR_MIN, FACTOR_MAX)
    factor = np.where(np.isfinite(factor), factor, 1.0)

    # Flags (FIX: numpy dtype, NOT "Int64")
    was_capped_pre = np.where((factor_raw < FACTOR_MIN) | (factor_raw > FACTOR_MAX), 1, 0).astype("int8")
    was_capped_post = np.where((factor < FACTOR_MIN) | (factor > FACTOR_MAX), 1, 0).astype("int8")

    # Output
    out = tmp[key].copy()
    out["factor_curva_ml1_raw"] = factor_raw
    out["factor_curva_ml1"] = factor
    out["ml1_version"] = ver_dir.name
    out["created_at"] = pd.Timestamp.utcnow()

    # Audit columns (keep names compatible with tus auditorías)
    out["_share_source"] = np.where(use_model, "cdf_adj", np.where(sb > 0, "baseline", "zero"))
    out["cap_share"] = np.nan  # placeholder (si luego metes cap/floor por bins)
    out["share_pred_in"] = tmp["share_pred_in"].to_numpy(dtype=float)
    out["share_smooth"] = tmp["share_smooth"].to_numpy(dtype=float)
    out["share_curva_ml1"] = tmp["share_curva_ml1"].to_numpy(dtype=float)
    out["tallos_pred_ml1_dia_smooth"] = tmp["tallos_pred_ml1_dia_from_cdf"].to_numpy(dtype=float)
    out["factor_curva_ml1_raw_smooth"] = factor_raw  # aquí factor_raw ya viene del share_smooth
    out["was_capped_pre"] = was_capped_pre
    out["was_capped_post"] = was_capped_post

    out = out.sort_values(["bloque_base", "variedad_canon", "fecha"]).reset_index(drop=True)
    write_parquet(out, OUT_PATH)
    print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name}")


if __name__ == "__main__":
    main(version=None)

------------------------------------------------------
[38/65] FILE: \src\models\ml1\apply_curva_share_dia.py
------------------------------------------------------
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet


# =============================================================================
# Paths
# =============================================================================
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")

REGISTRY_ROOT = Path("models_registry/ml1/curva_share_dia")

OUT_PATH = Path("data/gold/pred_factor_curva_ml1.parquet")
```

```
# ============================================================================
# Model columns (must match TRAIN)
# ============================================================================
NUM_COLS = [
    "day_in_harvest",
    "rel_pos",
    "n_harvest_days",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]

CAT_COLS = ["variedad_canon", "area", "tipo_sp"]


# ============================================================================
# Helpers
# ============================================================================
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    # OJO: pandas nullable Int64 (capital I) existe; numpy int64 no sirve si hay NA.
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")


def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)

    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base


def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df

    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)

    keep: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        if len(idxs) == 1:
            keep[c] = out.iloc[:, idxs[0]]
        else:
            s = out.iloc[:, idxs[0]]
            for j in idxs[1:]:
                s2 = out.iloc[:, j]
                s = s.where(s.notna(), s2)
            keep[c] = s

    ordered: list[str] = []
    for c in out.columns.astype(str):
        if c not in ordered:
            ordered.append(c)

    return pd.DataFrame({c: keep[c] for c in ordered})


def _infer_is_harvest(df: pd.DataFrame) -> np.ndarray:
    """
    Define máscara de días dentro de ventana harvest.
    Preferimos:
      - harvest_start_pred / harvest_end_pred (si existen)
      - day_in_harvest (si existe y es >=1)
      - fallback: baseline > 0
    """
    f = pd.to_datetime(df["fecha"], errors="coerce")

    hs_col = None
    he_col = None
    for c in ["harvest_start_pred", "harvest_start", "fecha_inicio_real"]:
        if c in df.columns:
            hs_col = c
            break
    for c in ["harvest_end_pred", "harvest_end_eff", "harvest_end", "fecha_fin_real"]:
        if c in df.columns:
            he_col = c
            break

    if hs_col and he_col:
        hs = pd.to_datetime(df[hs_col], errors="coerce")
        he = pd.to_datetime(df[he_col], errors="coerce")
        m = hs.notna() & he.notna() & f.notna() & (f >= hs) & (f <= he)
        return m.to_numpy(dtype=bool)

    if "day_in_harvest" in df.columns:
        dih = pd.to_numeric(df["day_in_harvest"], errors="coerce")
        m = dih.notna() & (dih.astype(float) >= 1)
        return m.to_numpy(dtype=bool)

    base = pd.to_numeric(df.get("tallos_pred_baseline_dia", 0.0), errors="coerce").fillna(0.0)
    return (base > 0).to_numpy(dtype=bool)


def _make_relpos_bin(rel_pos: pd.Series) -> pd.Series:
    """
    Bins 0..1 en 6 tramos (consistente con lo que te está saliendo: global rows=6).
    """
    x = pd.to_numeric(rel_pos, errors="coerce")
    bins = [-np.inf, 0.0, 0.10, 0.25, 0.50, 0.75, np.inf]
    labels = ["B0", "B1", "B2", "B3", "B4", "B5"]
    return pd.cut(x, bins=bins, labels=labels, include_lowest=True).astype("object")
```

```
def _safe_load_cap_tables(ver_dir: Path) -> tuple[pd.DataFrame | None, pd.DataFrame | None]:
    """
    Intenta cargar:
      - cap_floor_share_by_relpos.parquet (segmentado)
      - cap_floor_share_by_relpos_global.parquet (fallback global)
    desde la carpeta de versión.
    """
    seg = ver_dir / "cap_floor_share_by_relpos.parquet"
    glb = ver_dir / "cap_floor_share_by_relpos_global.parquet"
    seg_df = read_parquet(seg) if seg.exists() else None
    glb_df = read_parquet(glb) if glb.exists() else None
    return seg_df, glb_df


def _apply_cap_floor(
    df: pd.DataFrame,
    share_col: str,
    seg_caps: pd.DataFrame | None,
    glb_caps: pd.DataFrame | None,
) -> pd.DataFrame:
    """
    Aplica cap/floor por (variedad_canon, area, tipo_sp, rel_pos_bin) y fallback global por rel_pos_bin.
    No hardcodea ?punteos/colas?, solo limita extremos estadísticos.
    """
    out = df.copy()

    out[share_col] = pd.to_numeric(out[share_col], errors="coerce").fillna(0.0)
    out[share_col] = out[share_col].clip(lower=0.0)

    # preparar bins
    out["rel_pos_bin"] = _make_relpos_bin(out.get("rel_pos", np.nan))

    cap = pd.Series([np.nan] * len(out), index=out.index)
    floor = pd.Series([np.nan] * len(out), index=out.index)

    # segment caps
    if seg_caps is not None and len(seg_caps):
        s = seg_caps.copy()
        # normalizar nombres esperados mínimos
        for c in ["variedad_canon", "area", "tipo_sp"]:
            if c in s.columns:
                s[c] = _canon_str(s[c])
        if "rel_pos_bin" in s.columns:
            s["rel_pos_bin"] = s["rel_pos_bin"].astype("object")

        # detectar columnas cap/floor
        cap_col = "cap_share" if "cap_share" in s.columns else None
        floor_col = "floor_share" if "floor_share" in s.columns else None

        keys = [c for c in ["variedad_canon", "area", "tipo_sp", "rel_pos_bin"] if c in s.columns]
        if cap_col and keys:
            tmp = out.merge(
                s[keys + [cap_col] + ([floor_col] if floor_col else [])],
                on=keys,
                how="left",
                suffixes=("", "_cap"),
            )
            cap = pd.to_numeric(tmp[cap_col], errors="coerce")
            if floor_col:
                floor = pd.to_numeric(tmp[floor_col], errors="coerce")

    # global caps
    if glb_caps is not None and len(glb_caps):
        g = glb_caps.copy()
        if "rel_pos_bin" in g.columns:
            g["rel_pos_bin"] = g["rel_pos_bin"].astype("object")
        cap_col_g = "cap_share" if "cap_share" in g.columns else None
        floor_col_g = "floor_share" if "floor_share" in g.columns else None

        if cap_col_g and "rel_pos_bin" in g.columns:
            tmpg = out[["rel_pos_bin"]].merge(g[["rel_pos_bin", cap_col_g] + ([floor_col_g] if floor_col_g els
e [])],
                                             on="rel_pos_bin", how="left")
            cap_g = pd.to_numeric(tmpg[cap_col_g], errors="coerce")
            floor_g = pd.to_numeric(tmpg[floor_col_g], errors="coerce") if floor_col_g else pd.Series([np.nan]
*len(out), index=out.index)

            cap = cap.where(cap.notna(), cap_g)
            floor = floor.where(floor.notna(), floor_g)

    out["cap_share"] = cap
    out["floor_share"] = floor

    # aplicar
    was_cap = pd.Series(False, index=out.index)
    was_floor = pd.Series(False, index=out.index)

    if out["cap_share"].notna().any():
        m = out[share_col] > out["cap_share"]
        was_cap = was_cap | m.fillna(False)
        out.loc[m.fillna(False), share_col] = out.loc[m.fillna(False), "cap_share"]

    if out["floor_share"].notna().any():
        m = out[share_col] < out["floor_share"]
        was_floor = was_floor | m.fillna(False)
        out.loc[m.fillna(False), share_col] = out.loc[m.fillna(False), "floor_share"]

    out["was_capped"] = was_cap
    out["was_floored"] = was_floor
    return out


def _smooth_and_renorm(df: pd.DataFrame, share_col: str, win: int = 5) -> pd.Series:
    """
    Suaviza share por ciclo con rolling mean centrado y renormaliza a sum=1.
    No impone ?porcentajes fijos?; solo reduce picos espurios.
    """
    if win < 3:
        win = 3
    if win % 2 == 0:
        win += 1

    out = np.zeros(len(df), dtype=float)

    # asumimos df ya ordenado por ciclo/fecha
    grp = df.groupby("ciclo_id", dropna=False, sort=False)
    for _, sub in grp:
        idx = sub.index.to_numpy()
        s = pd.to_numeric(sub[share_col], errors="coerce").fillna(0.0).to_numpy(dtype=float)
        if len(s) == 0:
            continue
        # rolling mean centrado
        ss = pd.Series(s).rolling(window=win, center=True, min_periods=max(1, win // 2)).mean().to_numpy(dtype
=float)
        ss = np.clip(ss, 0.0, None)
        tot = float(np.nansum(ss))
        if tot > 0:
            ss = ss / tot
        out[idx] = ss

    return pd.Series(out, index=df.index, dtype=float)


# ==============================================================================
# Main
# ==============================================================================
def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    model_path = ver_dir / "model_curva_share_dia.joblib"
    if not metrics_path.exists():
        raise FileNotFoundError(f"No encontré metrics.json en {ver_dir}")
    if not model_path.exists():
        raise FileNotFoundError(f"No encontré modelo: {model_path}")

    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    feature_names = meta.get("feature_names")
    if not feature_names:
        raise ValueError("metrics.json no trae feature_names (necesario para alinear dummies en apply).")

    model = load(model_path)

    # caps
    seg_caps, glb_caps = _safe_load_cap_tables(ver_dir)

    # inputs
    feat = read_parquet(FEATURES_PATH).copy()
    feat = _dedupe_columns(feat)

    _require(
        feat,
        ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "tallos_proy"],
        "features_curva",
    )

    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["bloque_base"] = _canon_int(feat["bloque_base"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            feat[c] = _canon_str(feat[c])

    # alias posición
    _coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_fina
l"])
    _coalesce_cols(feat, "rel_pos", ["rel_pos", "rel_pos_pred", "rel_pos_pred_final"])
    _coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_fina
l"])

    # asegurar columnas
    for c in NUM_COLS:
        if c not in feat.columns:
            feat[c] = np.nan
    for c in CAT_COLS:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"

    # canon cats (importante antes de dummies)
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    feat["area"] = _canon_str(feat.get("area", "UNKNOWN")).fillna("UNKNOWN")
    feat["tipo_sp"] = _canon_str(feat.get("tipo_sp", "UNKNOWN")).fillna("UNKNOWN")

    # numerics
    for c in NUM_COLS:
        feat[c] = pd.to_numeric(feat[c], errors="coerce")

    feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna
(0.0).astype(float)
    feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0).astype(float)

    # universo
    uni = read_parquet(UNIVERSE_PATH).copy()
    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_ml1")
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key].drop_duplicates()

    # panel = universe LEFT join feat
    panel = uni_k.merge(feat, on=key, how="left", suffixes=("", "_feat"))

    # completar cats defaults si quedaron NaN (por universe sin match)
    panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
    panel["area"] = _canon_str(panel.get("area", "UNKNOWN")).fillna("UNKNOWN")
    panel["tipo_sp"] = _canon_str(panel.get("tipo_sp", "UNKNOWN")).fillna("UNKNOWN")

    # completar numerics faltantes
    for c in NUM_COLS:
        if c not in panel.columns:
            panel[c] = np.nan
        panel[c] = pd.to_numeric(panel[c], errors="coerce")

    panel["tallos_pred_baseline_dia"] = pd.to_numeric(panel.get("tallos_pred_baseline_dia", 0.0), errors="coer
ce").fillna(0.0).astype(float)
    panel["tallos_proy"] = pd.to_numeric(panel.get("tallos_proy", 0.0), errors="coerce").fillna(0.0).astype(fl
oat)

    # infer harvest mask
    is_h = _infer_is_harvest(panel)

    # X (dummies aligned)
    X = panel[NUM_COLS + CAT_COLS].copy()
    X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

    # IMPORTANT: align to training feature_names
    X = X.reindex(columns=feature_names, fill_value=0)

    # predict share raw
    share_pred = model.predict(X)
    share_pred = pd.to_numeric(pd.Series(share_pred, index=panel.index), errors="coerce").fillna(0.0).to_numpy
(dtype=float)
    share_pred = np.clip(share_pred, 0.0, None)
    share_pred = np.where(is_h, share_pred, 0.0)

    panel["share_pred_in"] = share_pred

    # cap/floor PRE
    tmp = panel.copy()
    tmp["_share_source"] = "model"
    tmp["share_curva_ml1_raw"] = tmp["share_pred_in"]

    tmp = _apply_cap_floor(tmp, "share_curva_ml1_raw", seg_caps, glb_caps)
    tmp["was_capped_pre"] = tmp["was_capped"]
    tmp["was_floored_pre"] = tmp["was_floored"]

    # renorm raw share por ciclo (si quedó todo 0, fallback baseline weights)
    base = tmp["tallos_pred_baseline_dia"].to_numpy(dtype=float)
    base_h = np.where(is_h, base, 0.0)

    # sum raw por ciclo
    sraw = tmp.groupby("ciclo_id", dropna=False)["share_curva_ml1_raw"].transform("sum").to_numpy(dtype=float)
    sb = pd.Series(base_h, index=tmp.index).groupby(tmp["ciclo_id"], dropna=False).transform("sum").to_numpy(d
type=float)

    share_raw = tmp["share_curva_ml1_raw"].to_numpy(dtype=float)
    share_ren = np.where(
        sraw > 0,
        share_raw / sraw,
        np.where(sb > 0, base_h / sb, 0.0),
    )
    tmp["share_curva_ml1"] = share_ren

    # smooth + renorm
    tmp = tmp.sort_values(["ciclo_id", "fecha"], kind="mergesort").reset_index(drop=True)
    tmp["share_smooth"] = _smooth_and_renorm(tmp, "share_curva_ml1", win=5)

    # cap/floor POST sobre share_smooth (opcional pero útil para evitar rebotes negativos/raros)
    tmp = _apply_cap_floor(tmp, "share_smooth", seg_caps, glb_caps)
    tmp["was_capped_post"] = tmp["was_capped"]
    tmp["was_floored_post"] = tmp["was_floored"]

    # renorm final post-cap
    s2 = tmp.groupby("ciclo_id", dropna=False)["share_smooth"].transform("sum").to_numpy(dtype=float)
    tmp["share_smooth"] = np.where(s2 > 0, tmp["share_smooth"].to_numpy(dtype=float) / s2, 0.0)

    # cyc_total = tallos_proy max por ciclo (fallback sum baseline)
    tproy_max = tmp.groupby("ciclo_id", dropna=False)["tallos_proy"].transform("max").to_numpy(dtype=float)
    tproy_max = np.where(np.isfinite(tproy_max), tproy_max, 0.0)
    cyc_total = np.where(tproy_max > 0, tproy_max, sb)

    tmp["tallos_pred_ml1_dia_smooth"] = cyc_total * tmp["share_smooth"].to_numpy(dtype=float)

    # factor compatible downstream
    eps = 1e-9
    tmp["factor_curva_ml1_raw_smooth"] = np.where(
        is_h,
        tmp["tallos_pred_ml1_dia_smooth"].to_numpy(dtype=float) / (base + eps),
        1.0,
    )

    FACTOR_MIN, FACTOR_MAX = 0.2, 5.0
    tmp["factor_curva_ml1_raw"] = tmp["factor_curva_ml1_raw_smooth"]
    tmp["factor_curva_ml1"] = np.clip(tmp["factor_curva_ml1_raw"].to_numpy(dtype=float), FACTOR_MIN, FACTOR_MA
X)
    tmp["factor_curva_ml1"] = np.where(np.isfinite(tmp["factor_curva_ml1"]), tmp["factor_curva_ml1"], 1.0)

    # metadata
    tmp["ml1_version"] = ver_dir.name
    tmp["created_at"] = pd.Timestamp.utcnow()

    # outputs
    out_cols = [
        "ciclo_id",
        "fecha",
        "bloque_base",
        "variedad_canon",
        "factor_curva_ml1",
        "factor_curva_ml1_raw",
        "ml1_version",
        "created_at",
        "_share_source",
        "cap_share",
        "share_pred_in",
        "share_smooth",
        "share_curva_ml1",
        "tallos_pred_ml1_dia_smooth",
        "factor_curva_ml1_raw_smooth",
        "was_capped_pre",
        "was_capped_post",
        "was_floored_pre",
        "was_floored_post",
    ]
    out_cols = [c for c in out_cols if c in tmp.columns]

    out = tmp[out_cols].sort_values(["bloque_base", "variedad_canon", "fecha"], kind="mergesort").reset_index(
drop=True)

    write_parquet(out, OUT_PATH)

    # quick audits
    # share sum by cycle
    ss = out.groupby("ciclo_id", dropna=False)["share_smooth"].sum()
    # mass balance
    tallos_sum = out.groupby("ciclo_id", dropna=False)["tallos_pred_ml1_dia_smooth"].sum()
    proy = tmp.groupby("ciclo_id", dropna=False)["tallos_proy"].max()
    max_abs = float((tallos_sum - proy).abs().max()) if len(tallos_sum) else float("nan")

    print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name}")
    print(f"[CHECK] share_smooth sum min/max: {float(ss.min()):.6f} / {float(ss.max()):.6f}")
    print(f"[CHECK] ciclo mass-balance ML1 vs tallos_proy | max abs diff: {max_abs:.12f}")
```

```
if __name__ == "__main__":
    main(version=None)

-------------------------------------------------------
[39/65] FILE: \src\models\ml1\apply_curva_tallos_dia.py
-------------------------------------------------------
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet


FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/curva_tallos_dia")
OUT_PATH = Path("data/gold/pred_factor_curva_ml1.parquet")


NUM_COLS = [
    "tallos_pred_baseline_dia",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]

CAT_COLS = ["variedad_canon", "area", "tipo_sp"]


def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]


def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    model_path = ver_dir / "model_curva_tallos_dia.joblib"
    if not model_path.exists():
        raise FileNotFoundError(f"No encontré modelo: {model_path}")

    model = load(model_path)

    df = read_parquet(FEATURES_PATH).copy()
    need = {"fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"FEATURES curva sin columnas necesarias: {sorted(miss)}")
    # después de leer df
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    if "ciclo_id" in df.columns:
        df["ciclo_id"] = df["ciclo_id"].astype(str)
    df["bloque_base"] = pd.to_numeric(df["bloque_base"], errors="coerce").astype("Int64")
    df["variedad_canon"] = df["variedad_canon"].astype(str).str.upper().str.strip()

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    if df.duplicated(subset=key).any():
        df = df.drop_duplicates(subset=key, keep="first")

    # asegurar columnas
    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    X = df[NUM_COLS + CAT_COLS]
    pred = model.predict(X)

    out = df[["fecha", "bloque_base", "variedad_canon"]].copy()
    if "ciclo_id" in df.columns:
        out["ciclo_id"] = df["ciclo_id"]

    out["ml1_version"] = ver_dir.name
    out["factor_curva_ml1_raw"] = pd.to_numeric(pred, errors="coerce")

    # safety clip (evita volar planificación)
    out["factor_curva_ml1"] = out["factor_curva_ml1_raw"].clip(lower=0.2, upper=5.0)

    # fallback si quedó NaN por cualquier razón
    out["factor_curva_ml1"] = out["factor_curva_ml1"].fillna(1.0)

    out["created_at"] = pd.Timestamp.utcnow()

    cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "factor_curva_ml1", "factor_curva_ml1_raw",
"ml1_version", "created_at"]
    cols = [c for c in cols if c in out.columns]
    out = out[cols].sort_values(["bloque_base", "variedad_canon", "fecha"]).reset_index(drop=True)

    write_parquet(out, OUT_PATH)
    print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name}")
    print(f"best_model: {meta.get('best_model')}")


if __name__ == "__main__":
    main(version=None)

------------------------------------------------------------
[40/65] FILE: \src\models\ml1\train_ajuste_poscosecha_ml1.py
------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

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

```python
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

DIM_PATH = Path("data/silver/dim_mermas_ajuste_fecha_post_destino.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/ajuste_poscosecha")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino"]


def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )
    pre = ColumnTransformer(
        transformers=[("num", num_pipe, NUM_COLS), ("cat", cat_pipe, CAT_COLS)],
        remainder="drop",
        sparse_threshold=0.0,
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])


def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0, random_state=0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }


def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize().dropna()
    if d.empty:
        return []
    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []
    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]
    folds = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        vs, ve = vb.min(), vb.max()
        tr = (all_dates < vs).to_numpy()
        va = ((all_dates >= vs) & (all_dates <= ve)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))
    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds


def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    dim = read_parquet(DIM_PATH).copy()
    dim.columns = [str(c).strip() for c in dim.columns]

    need = {"fecha_post", "destino", "ajuste"}
    miss = need - set(dim.columns)
    if miss:
        raise ValueError(f"dim_mermas_ajuste_fecha_post_destino sin columnas: {sorted(miss)}")

    df = dim.copy()
    df["fecha_post"] = pd.to_datetime(df["fecha_post"], errors="coerce").dt.normalize()
    df["destino"] = df["destino"].astype(str).str.upper().str.strip()
    df["y"] = pd.to_numeric(df["ajuste"], errors="coerce")

    df = df[df["fecha_post"].notna() & df["destino"].notna() & df["y"].notna()].copy()
    df["y"] = df["y"].clip(lower=0.80, upper=1.50)  # <- evita el ruido del 0.5 ?cap extremo?

    if len(df) < 100:
        raise ValueError(f"Poca data para entrenar ajuste (n={len(df)}).")

    df["dow"] = df["fecha_post"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha_post"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha_post"].dt.isocalendar().week.astype("Int64")

    X = df[NUM_COLS + CAT_COLS].copy()
    y = df["y"].astype(float).to_numpy()

    folds = _time_folds(df["fecha_post"], n_folds=4)
    if not folds:
        raise ValueError("No pude armar folds temporales para ajuste.")

    models = _candidate_models()

    best_name, best_score, best_model = None, None, None
    all_metrics = {}

    for name, model in models.items():
        pipe = _make_pipeline(model)
        maes = []
        for tr_idx, va_idx in folds:
            pipe.fit(X.iloc[tr_idx], y[tr_idx])
            pred = pipe.predict(X.iloc[va_idx])
            maes.append(float(np.mean(np.abs(y[va_idx] - pred))))
        mae_mean = float(np.mean(maes))
        mae_std = float(np.std(maes))
        score = 0.85 * mae_mean + 0.15 * mae_std

        all_metrics[name] = {"mae_mean": mae_mean, "mae_std": mae_std, "score": float(score),
                             "n_rows": int(len(X)), "n_folds": int(len(folds))}
        if (best_score is None) or (score < best_score):
            best_score, best_name, best_model = score, name, model

    best_pipe = _make_pipeline(best_model)
    best_pipe.fit(X, y)

    model_path = out_dir / "model.joblib"
    dump(best_pipe, model_path)

    summary = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "dim_path": str(DIM_PATH).replace("\\", "/"),
        "target": "ajuste",
        "clip_range_apply": [0.80, 1.50],
        "features": {"num": NUM_COLS, "cat": CAT_COLS},
        "best_model": best_name,
        "metrics": all_metrics,
        "model_path": str(model_path).replace("\\", "/"),
    }

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {out_dir}/ (model.joblib + metrics.json) | best={best_name} score={best_score:.6f}")


if __name__ == "__main__":
    main()

----------------------------------------------------------------
[41/65] FILE: \src\models\ml1\train_curva_beta_multiplier_dia.py
----------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import dump
from sklearn.ensemble import HistGradientBoostingRegressor

from common.io import read_parquet

TRAINSET_PATH = Path("data/features/trainset_curva_beta_multiplier_dia.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/curva_beta_multiplier_dia")

NUM_COLS = [
    "day_in_harvest","rel_pos","n_harvest_days",
    "pct_avance_real","dia_rel_cosecha_real","gdc_acum_real",
    "rainfall_mm_dia","horas_lluvia","temp_avg_dia","solar_energy_j_m2_dia",
    "wind_speed_avg_dia","wind_run_dia","gdc_dia",
    "dias_desde_sp","gdc_acum_desde_sp",
    "dow","month","weekofyear",
]
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]

def _make_version() -> str:
    return pd.Timestamp.utcnow().strftime("%Y%m%d_%H%M%S")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    if not TRAINSET_PATH.exists():
        raise FileNotFoundError(f"No existe: {TRAINSET_PATH}")

    created_at = pd.Timestamp.utcnow()
    version = _make_version()
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(TRAINSET_PATH).copy()

    # Ensure cols exist
    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    for c in CAT_COLS:
        df[c] = _canon_str(df[c].fillna("UNKNOWN"))

    y = pd.to_numeric(df["y_log_mult"], errors="coerce")
    ok = y.notna()
    df = df[ok].copy()
    y = y[ok].astype(float)

    # weights: dar más peso a días con mayor real (para que aprenda supresión por clima también se puede)
    if "tallos_real_dia" in df.columns:
        w = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
        w = np.sqrt(np.clip(w, 0.0, None))
        w = np.where(w > 0, w / np.median(w[w > 0]), 1.0)
        w = np.clip(w, 0.2, 5.0)
    else:
        w = np.ones(len(df), dtype=float)

    # X
    X = df[NUM_COLS + CAT_COLS].copy()
    for c in NUM_COLS:
        X[c] = pd.to_numeric(X[c], errors="coerce")
    X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

    model = HistGradientBoostingRegressor(
        loss="squared_error",
        max_depth=6,
        learning_rate=0.06,
        max_leaf_nodes=31,
        min_samples_leaf=120,     # más alto para suavidad
        l2_regularization=1e-4,
        random_state=42,
    )
    model.fit(X, y.to_numpy(dtype=float), sample_weight=w)

    dump(model, out_dir / "model_log_mult.joblib")

    meta = {
        "created_at": str(created_at),
        "version": version,
        "target": "y_log_mult = log(share_real/(share_beta+eps)) (daily multiplier on beta prior)",
        "feature_cols_numeric": NUM_COLS,
        "feature_cols_categorical": CAT_COLS,
        "feature_names": list(X.columns),
        "n_train_rows": int(len(X)),
        "clip_note": "apply will clip multiplier exp(y) to stable bounds",
    }
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(meta, f, indent=2, ensure_ascii=False)

    print(f"OK -> {out_dir} | n_train_rows={len(X):,}")

if __name__ == "__main__":
    main()

--------------------------------------------------------
[42/65] FILE: \src\models\ml1\train_curva_beta_params.py
--------------------------------------------------------
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import dump
from sklearn.ensemble import HistGradientBoostingRegressor

from common.io import read_parquet

TRAINSET_PATH = Path("data/features/trainset_curva_beta_params.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/curva_beta_params")

# Columnas categóricas
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]

# Targets (alpha, beta) siempre > 1
# Modelamos z = log(target - 1) para asegurar positividad al aplicar: target = 1 + exp(z)
TARGET_ALPHA = "alpha"
```

```python
TARGET_BETA = "beta"

def _make_version() -> str:
    return pd.Timestamp.utcnow().strftime("%Y%m%d_%H%M%S")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    if not TRAINSET_PATH.exists():
        raise FileNotFoundError(f"No existe: {TRAINSET_PATH}")

    created_at = pd.Timestamp.utcnow()
    version = _make_version()
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(TRAINSET_PATH).copy()

    # Canon cats
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"
        df[c] = _canon_str(df[c].fillna("UNKNOWN"))

    # targets
    df[TARGET_ALPHA] = pd.to_numeric(df[TARGET_ALPHA], errors="coerce")
    df[TARGET_BETA] = pd.to_numeric(df[TARGET_BETA], errors="coerce")

    ok = df[TARGET_ALPHA].notna() & df[TARGET_BETA].notna()
    df = df[ok].copy()
    if len(df) < 50:
        raise ValueError(f"Trainset muy pequeño ({len(df)}). Revisa build_targets o MIN_REAL_TOTAL.")

    # Features numéricas: todo excepto ids/cats/targets
    drop_cols = {"ciclo_id", "bloque_base", "created_at", TARGET_ALPHA, TARGET_BETA}
    num_cols = [c for c in df.columns if c not in drop_cols and c not in CAT_COLS]
    # limpia
    for c in num_cols:
        df[c] = pd.to_numeric(df[c], errors="coerce")

    X = df[num_cols + CAT_COLS].copy()
    X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

    # Targets transformados
    y_a = np.log(np.clip(df[TARGET_ALPHA].to_numpy(dtype=float) - 1.0, 1e-6, None))
    y_b = np.log(np.clip(df[TARGET_BETA].to_numpy(dtype=float) - 1.0, 1e-6, None))

    # sample_weight: más peso a ciclos con mayor real_total si existe
    if "real_total" in df.columns:
        w = pd.to_numeric(df["real_total"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
        w = np.sqrt(np.clip(w, 0.0, None))
        w = np.where(w > 0, w / np.median(w[w > 0]), 1.0)
        w = np.clip(w, 0.2, 5.0)
    else:
        w = np.ones(len(df), dtype=float)

    # Modelos
    model_a = HistGradientBoostingRegressor(
        loss="squared_error",
        max_depth=6,
        learning_rate=0.06,
        max_leaf_nodes=31,
        min_samples_leaf=60,
        l2_regularization=1e-4,
        random_state=42,
    )
    model_b = HistGradientBoostingRegressor(
        loss="squared_error",
        max_depth=6,
        learning_rate=0.06,
        max_leaf_nodes=31,
        min_samples_leaf=60,
        l2_regularization=1e-4,
        random_state=42,
    )

    model_a.fit(X, y_a, sample_weight=w)
    model_b.fit(X, y_b, sample_weight=w)

    dump(model_a, out_dir / "model_alpha.joblib")
    dump(model_b, out_dir / "model_beta.joblib")

    meta = {
        "created_at": str(created_at),
        "version": version,
        "target": "alpha,beta params of Beta PDF on rel_pos; trained on real harvest shares (cycle-level)",
        "transform": "z = log(param - 1), param = 1 + exp(z) (ensures >1 unimodal)",
        "feature_cols_numeric": num_cols,
        "feature_cols_categorical": CAT_COLS,
        "feature_names": list(X.columns),
        "n_train_rows": int(len(X)),
        "n_groups": int(df[["ciclo_id", "bloque_base", "variedad_canon"]].drop_duplicates().shape[0]),
    }
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(meta, f, indent=2, ensure_ascii=False)

    print(f"OK -> {out_dir} | n_train_rows={len(X):,} | n_groups={meta['n_groups']:,}")

if __name__ == "__main__":
    main()

----------------------------------------------------
[43/65] FILE: \src\models\ml1\train_curva_cdf_dia.py
----------------------------------------------------
from __future__ import annotations

from pathlib import Path
import json

import numpy as np
import pandas as pd
from joblib import dump
from sklearn.ensemble import HistGradientBoostingRegressor

from common.io import read_parquet

FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")

REGISTRY_ROOT = Path("models_registry/ml1/curva_cdf_dia")

# Inputs numéricos (si faltan, se crean como NaN)
NUM_COLS = [
    "day_in_harvest",
    "rel_pos",
    "n_harvest_days",
    "pct_avance_real",          # opcional
    "dia_rel_cosecha_real",     # opcional
    "gdc_acum_real",            # opcional
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]

CAT_COLS = ["variedad_canon", "area", "tipo_sp"]
CAT_COLS_MERGE = ["area", "tipo_sp"]


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")


def _make_version() -> str:
    # reemplazo de utcnow (deprecated)
    return pd.Timestamp.now("UTC").strftime("%Y%m%d_%H%M%S")


def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)
    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base


def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df

    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)

    keep_series: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        if len(idxs) == 1:
            keep_series[c] = out.iloc[:, idxs[0]]
        else:
            s = out.iloc[:, idxs[0]]
            for j in idxs[1:]:
                s2 = out.iloc[:, j]
                s = s.where(s.notna(), s2)
            keep_series[c] = s

    ordered: list[str] = []
    for c in out.columns.astype(str):
        if c not in ordered:
            ordered.append(c)

    return pd.DataFrame({c: keep_series[c] for c in ordered})


def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))


def main() -> None:
    for p in [FEATURES_PATH, UNIVERSE_PATH, PROG_PATH, DIM_VAR_PATH]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    created_at = pd.Timestamp.now("UTC")
    version = _make_version()
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    # ---- var map (PROG raw -> canon)
    dim_var = read_parquet(DIM_VAR_PATH).copy()
    var_map = _load_var_map(dim_var)

    # ---- features (baseline + clima + term + pred window)
    feat = _dedupe_columns(read_parquet(FEATURES_PATH).copy())
    if not pd.Index(feat.columns.astype(str)).is_unique:
        dup = pd.Index(feat.columns.astype(str))[pd.Index(feat.columns.astype(str)).duplicated()].unique().tol
ist()
        raise ValueError(f"features_curva aún tiene columnas duplicadas: {dup}")

    _require(
        feat,
        ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "tallos_proy"],
        "features_curva",
    )

    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["bloque_base"] = _canon_int(feat["bloque_base"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            feat[c] = _canon_str(feat[c])

    # aliases desde *_pred
    _coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_fina
l"])
    _coalesce_cols(feat, "rel_pos", ["rel_pos", "rel_pos_pred", "rel_pos_pred_final"])
    _coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_fina
l"])

    for c in ["day_in_harvest", "rel_pos", "n_harvest_days"]:
        feat[c] = pd.to_numeric(feat[c], errors="coerce")

    # asegurar columnas
    for c in NUM_COLS:
        if c not in feat.columns:
            feat[c] = np.nan
    for c in CAT_COLS:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"

    feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna
(0.0)
    feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0)

    # ---- universe
    uni = read_parquet(UNIVERSE_PATH).copy()
    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_ml1")
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key].drop_duplicates()

    # ---- progreso real (panelizado + zeros)
    prog = read_parquet(PROG_PATH).copy()
    _require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "dim_cosecha_progress_
bloque_fecha")
    prog["ciclo_id"] = prog["ciclo_id"].astype(str)
    prog["fecha"] = _to_date(prog["fecha"])
    prog["bloque_base"] = _canon_int(prog["bloque_base"])
    prog["variedad_raw"] = _canon_str(prog["variedad"])
    prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])
    prog["variedad_canon"] = _canon_str(prog["variedad_canon"])
    prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

    prog_k_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_real_dia"]
    if "pct_avance_real" in prog.columns:
        prog_k_cols.append("pct_avance_real")
    prog_k = prog[prog_k_cols].drop_duplicates(subset=key)

    feat_take = key + ["tallos_pred_baseline_dia", "tallos_proy"] + NUM_COLS + CAT_COLS_MERGE
    feat_take = list(dict.fromkeys(feat_take))

    panel = (
        uni_k
        .merge(feat[feat_take], on=key, how="left")
        .merge(prog_k, on=key, how="left")
    )

    # fill 0 en días no registrados
    panel["tallos_real_dia"] = pd.to_numeric(panel["tallos_real_dia"], errors="coerce").fillna(0.0)

    # asegurar features
    for c in NUM_COLS:
        if c not in panel.columns:
            panel[c] = np.nan
    for c in CAT_COLS:
        if c not in panel.columns:
            panel[c] = "UNKNOWN"

    panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
    if "area" in panel.columns:
        panel["area"] = _canon_str(panel["area"].fillna("UNKNOWN"))
    if "tipo_sp" in panel.columns:
        panel["tipo_sp"] = _canon_str(panel["tipo_sp"].fillna("UNKNOWN"))

    # ---- train cycles con señal real
    cyc_sum_real = panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
    train = panel[cyc_sum_real > 0].copy()

    # ---- máscara harvest (si no hay day_in_harvest, igual entrena con rel_pos; pero preferimos day_in_harves
t)
    dih = pd.to_numeric(train["day_in_harvest"], errors="coerce")
    nh = pd.to_numeric(train["n_harvest_days"], errors="coerce")
    is_h = dih.notna() & nh.notna() & (dih >= 1) & (nh >= 1) & (dih <= nh)
    train.loc[~is_h, "tallos_real_dia"] = 0.0

    # ---- target: CDF real por ciclo (ordenado por day_in_harvest; fallback a fecha si dih falta)
    train["_dih_sort"] = pd.to_numeric(train["day_in_harvest"], errors="coerce")

    # FIX PANDAS: Series.view ya no existe -> usar astype("int64") en datetime64[ns]
    # (fecha es datetime64[ns] por _to_date)
    fecha_int64 = pd.to_datetime(train["fecha"], errors="coerce").astype("int64")

    train["_sort_key"] = np.where(
        train["_dih_sort"].notna().to_numpy(),
        train["_dih_sort"].astype("float64").to_numpy(),
        fecha_int64.astype("float64").to_numpy(),
    )

    train = train.sort_values(["ciclo_id", "_sort_key"], kind="mergesort")

    tot = train.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
    cum = train.groupby("ciclo_id", dropna=False)["tallos_real_dia"].cumsum().astype(float)
    train["cdf_real"] = np.where(tot > 0, cum / tot, np.nan)

    # ---- sample_weight: si existe pct_avance_real, ramp 2%..5%; si no, usar cdf_real como proxy
    if "pct_avance_real" in train.columns and train["pct_avance_real"].notna().any():
        pav = pd.to_numeric(train["pct_avance_real"], errors="coerce").fillna(0.0).astype(float)
        sw = np.clip((pav - 0.02) / 0.03, 0.2, 1.0)
    else:
        cdfp = pd.to_numeric(train["cdf_real"], errors="coerce").fillna(0.0).astype(float)
        sw = np.clip((cdfp - 0.02) / 0.03, 0.2, 1.0)
    train["sample_weight"] = sw

    # ---- X/y
    y = pd.to_numeric(train["cdf_real"], errors="coerce")
    X = train[NUM_COLS + CAT_COLS].copy()

    ok = y.notna()
    X = X.loc[ok].copy()
    y = y.loc[ok].astype(float)
    sample_weight = train.loc[ok, "sample_weight"].astype(float).to_numpy()

    # one-hot + persist feature_names
    X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

    model = HistGradientBoostingRegressor(
        loss="squared_error",
        max_depth=6,
        learning_rate=0.08,
        max_leaf_nodes=31,
        min_samples_leaf=80,
        l2_regularization=1e-4,
        random_state=42,
    )
    model.fit(X, y, sample_weight=sample_weight)

    dump(model, out_dir / "model_curva_cdf_dia.joblib")

    meta = {
        "created_at": str(created_at),
        "version": version,
        "best_model": "HistGradientBoostingRegressor",
        "target": "cdf_real per-cycle (panelized on universe; missing prog days filled with 0; monotonic enfor
ced at apply)",
        "position_cols": "coalesce day_in_harvest_pred/rel_pos_pred/n_harvest_days_pred -> day_in_harvest/rel_
pos/n_harvest_days",
        "gating": "sample_weight ramps between 2%..5% using pct_avance_real if exists else cdf_real proxy",
        "feature_cols_numeric": NUM_COLS,
        "feature_cols_categorical": CAT_COLS,
        "feature_names": list(X.columns),
        "n_train_rows": int(len(X)),
        "n_cycles_train": int(train.loc[ok, "ciclo_id"].nunique()),
    }
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(meta, f, indent=2, ensure_ascii=False)

    print(f"OK -> {out_dir} | n_train_rows={len(X):,} | n_cycles={meta['n_cycles_train']:,}")


if __name__ == "__main__":
    main()

------------------------------------------------------
[44/65] FILE: \src\models\ml1\train_curva_share_dia.py
------------------------------------------------------
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import dump
from sklearn.ensemble import HistGradientBoostingRegressor
```

```python
from common.io import read_parquet


FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")

REGISTRY_ROOT = Path("models_registry/ml1/curva_share_dia")


# -------------------------
# Model columns
# -------------------------
NUM_COLS = [
    "day_in_harvest",
    "rel_pos",
    "n_harvest_days",
    "pct_avance_real",          # <- opcional en data, pero el modelo la puede usar si existe
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]


# -------------------------
# Helpers
# -------------------------
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")


def _make_version() -> str:
    return pd.Timestamp.utcnow().strftime("%Y%m%d_%H%M%S")


def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df

    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)
    keep: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        s = out.iloc[:, idxs[0]]
        for j in idxs[1:]:
            s2 = out.iloc[:, j]
            s = s.where(s.notna(), s2)
        keep[c] = s

    ordered: list[str] = []
    for c in out.columns.astype(str):
        if c not in ordered:
            ordered.append(c)

    return pd.DataFrame({c: keep[c] for c in ordered})


def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))


def _relpos_bin(rel_pos: pd.Series) -> pd.Series:
    x = pd.to_numeric(rel_pos, errors="coerce").astype(float)
    bins = [-np.inf, 0.05, 0.15, 0.30, 0.60, 0.85, np.inf]
    labels = ["00-05", "05-15", "15-30", "30-60", "60-85", "85-100"]
    return pd.cut(x, bins=bins, labels=labels)


def _ndays_bucket(n: pd.Series) -> pd.Series:
    x = pd.to_numeric(n, errors="coerce")
    return pd.cut(
        x.astype(float),
        bins=[-np.inf, 30, 45, 60, np.inf],
        labels=["<=30", "31-45", "46-60", ">60"],
    )


def _safe_quantile(s: pd.Series, q: float) -> float:
    v = pd.to_numeric(s, errors="coerce").dropna().astype(float)
    if len(v) < 30:
        return float("nan")
    return float(v.quantile(q))


def main() -> None:
    for p in [FEATURES_PATH, UNIVERSE_PATH, PROG_PATH, DIM_VAR_PATH]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    created_at = pd.Timestamp.utcnow()
    version = _make_version()
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    # -------------------------
    # Dim variedad canon
    # -------------------------
    dim_var = read_parquet(DIM_VAR_PATH).copy()
    var_map = _load_var_map(dim_var)

    # -------------------------
    # Features
    # -------------------------
    feat = read_parquet(FEATURES_PATH).copy()
    feat = _dedupe_columns(feat)

    _require(
        feat,
        ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "tallos_proy"],
        "features_curva",
    )

    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["bloque_base"] = _canon_int(feat["bloque_base"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            feat[c] = _canon_str(feat[c])

    # asegurar columnas del modelo (num+cat) en FEAT
    for c in NUM_COLS:
        if c not in feat.columns:
            feat[c] = np.nan
    for c in CAT_COLS:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"

    feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna
(0.0)
    feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0)

    # -------------------------
    # Universe (positional)
    # -------------------------
    uni = read_parquet(UNIVERSE_PATH).copy()
    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "rel_pos_pred", "n_harvest_days_pred"
], "universe_harvest_grid_ml1")

    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key + ["rel_pos_pred", "day_in_harvest_pred", "n_harvest_days_pred"]].drop_duplicates(subset=k
ey)

    # -------------------------
    # Prog real (solo real + opcional pct_avance_real)
    # -------------------------
    prog = read_parquet(PROG_PATH).copy()
    _require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "dim_cosecha_progress_
bloque_fecha")

    prog["ciclo_id"] = prog["ciclo_id"].astype(str)
    prog["fecha"] = _to_date(prog["fecha"])
    prog["bloque_base"] = _canon_int(prog["bloque_base"])

    prog["variedad_raw"] = _canon_str(prog["variedad"])
    prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])

    prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

    prog_take = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_real_dia"]
    if "pct_avance_real" in prog.columns:
        prog_take.append("pct_avance_real")
    prog_k = prog[prog_take].drop_duplicates(subset=key)

    # -------------------------
    # Panel
    # -------------------------
    # Importante: traer SOLO lo que necesitamos de feat (evita duplicados raros)
    feat_take = key + ["tallos_pred_baseline_dia", "tallos_proy"] + NUM_COLS + ["area", "tipo_sp"]
    feat_take = [c for c in dict.fromkeys(feat_take) if c in feat.columns]

    panel = (
        uni_k.merge(feat[feat_take], on=key, how="left")
            .merge(prog_k, on=key, how="left")
    )

    # Asegurar TODAS las columnas del modelo en el panel (incl. pct_avance_real)
    for c in NUM_COLS:
        if c not in panel.columns:
            panel[c] = np.nan
    for c in CAT_COLS:
        if c not in panel.columns:
            panel[c] = "UNKNOWN"

    # Missing prog day => 0 real
    panel["tallos_real_dia"] = pd.to_numeric(panel["tallos_real_dia"], errors="coerce").fillna(0.0)

    # Position: usar *_pred como eje de forma
    panel["rel_pos"] = pd.to_numeric(panel["rel_pos"], errors="coerce")
    panel["rel_pos"] = panel["rel_pos"].where(panel["rel_pos"].notna(), pd.to_numeric(panel["rel_pos_pred"], e
rrors="coerce"))

    panel["day_in_harvest"] = pd.to_numeric(panel["day_in_harvest"], errors="coerce")
    panel["day_in_harvest"] = panel["day_in_harvest"].where(panel["day_in_harvest"].notna(), pd.to_numeric(pan
el["day_in_harvest_pred"], errors="coerce"))

    panel["n_harvest_days"] = pd.to_numeric(panel["n_harvest_days"], errors="coerce")
    panel["n_harvest_days"] = panel["n_harvest_days"].where(panel["n_harvest_days"].notna(), pd.to_numeric(pan
el["n_harvest_days_pred"], errors="coerce"))

    # Canon categóricas
    panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
    panel["area"] = _canon_str(panel.get("area", "UNKNOWN").fillna("UNKNOWN"))
    panel["tipo_sp"] = _canon_str(panel.get("tipo_sp", "UNKNOWN").fillna("UNKNOWN"))

    # -------------------------
    # Train set
    # -------------------------
    cyc_sum = panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
    train = panel[cyc_sum > 0].copy()

    denom = train.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
    train["share_real"] = np.where(denom > 0, train["tallos_real_dia"].astype(float) / denom, np.nan)

    # sample_weight por avance si existe; si no, weight=1
    pav = pd.to_numeric(train.get("pct_avance_real", np.nan), errors="coerce").fillna(0.0).astype(float)
    sw = np.ones(len(train), dtype=float)
    if "pct_avance_real" in train.columns:
        sw = np.clip((pav - 0.02) / 0.03, 0.2, 1.0)
    train["sample_weight"] = sw

    # X/y
    X = train[NUM_COLS + CAT_COLS].copy()
    y = pd.to_numeric(train["share_real"], errors="coerce")
    ok = y.notna()

    X = X.loc[ok].copy()
    y = y.loc[ok].astype(float)
    sample_weight = train.loc[ok, "sample_weight"].astype(float).to_numpy()

    # one-hot + guardar feature names
    X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

    model = HistGradientBoostingRegressor(
        loss="squared_error",
        max_depth=6,
        learning_rate=0.08,
        max_leaf_nodes=31,
        min_samples_leaf=80,
        l2_regularization=1e-4,
        random_state=42,
    )
    model.fit(X, y, sample_weight=sample_weight)
    dump(model, out_dir / "model_curva_share_dia.joblib")

    # -------------------------
    # Calibration caps/floors (learned)
    # -------------------------
    cal = train[["ciclo_id", "variedad_canon", "area", "tipo_sp", "rel_pos", "n_harvest_days", "share_real"]].
copy()
    cal["rel_pos_bin"] = _relpos_bin(cal["rel_pos"])
    cal["n_days_bucket"] = _ndays_bucket(cal["n_harvest_days"])

    seg_cols = ["variedad_canon", "area", "tipo_sp", "n_days_bucket", "rel_pos_bin"]
    g = cal.groupby(seg_cols, dropna=False)["share_real"]

    cap = g.apply(lambda s: _safe_quantile(s, 0.99)).rename("cap_share_p99")
    flo = g.apply(lambda s: _safe_quantile(s, 0.01)).rename("floor_share_p01")
    cal_tab = pd.concat([cap, flo], axis=1).reset_index()

    g2 = cal.groupby(["rel_pos_bin"], dropna=False)["share_real"]
    cal_glob = pd.DataFrame({
        "rel_pos_bin": g2.apply(lambda s: _safe_quantile(s, 0.99)).index.astype(str),
        "cap_share_p99_global": g2.apply(lambda s: _safe_quantile(s, 0.99)).to_numpy(),
        "floor_share_p01_global": g2.apply(lambda s: _safe_quantile(s, 0.01)).to_numpy(),
    })

    cal_tab["created_at"] = created_at
    cal_glob["created_at"] = created_at

    cal_tab_path = out_dir / "cap_floor_share_by_relpos.parquet"
    cal_glob_path = out_dir / "cap_floor_share_by_relpos_global.parquet"
    cal_tab.to_parquet(cal_tab_path, index=False)
    cal_glob.to_parquet(cal_glob_path, index=False)

    meta = {
        "created_at": str(created_at),
        "version": version,
        "best_model": "HistGradientBoostingRegressor",
        "target": "share_real per-cycle (panelized on universe; missing prog days filled with 0)",
        "feature_cols_numeric": NUM_COLS,
        "feature_cols_categorical": CAT_COLS,
        "feature_names": list(X.columns),
        "n_train_rows": int(len(X)),
        "n_cycles_train": int(train["ciclo_id"].nunique()),
        "calibration": {
            "rel_pos_bins": ["00-05", "05-15", "15-30", "30-60", "60-85", "85-100"],
            "segmentation": seg_cols,
            "cap_quantile": 0.99,
            "floor_quantile": 0.01,
            "min_samples_quantile": 30,
        },
    }
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(meta, f, indent=2, ensure_ascii=False)

    print(f"OK -> {out_dir} | n_train_rows={len(X):,} | n_cycles={meta['n_cycles_train']:,}")
    print(f"OK -> {cal_tab_path.name} | rows={len(cal_tab):,}")
    print(f"OK -> {cal_glob_path.name} | rows={len(cal_glob):,}")


if __name__ == "__main__":
    main()

-------------------------------------------------------
[45/65] FILE: \src\models\ml1\train_curva_tallos_dia.py
-------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")


FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/curva_tallos_dia")


NUM_COLS = [
    # baseline
    "tallos_pred_baseline_dia",
    # progreso / etapa
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    # clima / gdc
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    # SP
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    # calendario
    "dow",
    "month",
    "weekofyear",
]

CAT_COLS = [
    "variedad_canon",
    "area",
    "tipo_sp",
]


def _wape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = np.sum(np.abs(y_true))
    if denom <= 1e-12:
        return float(np.nan)
    return float(np.sum(np.abs(y_true - y_pred)) / denom)


def _smape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = (np.abs(y_true) + np.abs(y_pred))
    denom = np.where(denom < 1e-12, 1e-12, denom)
    return float(np.mean(2.0 * np.abs(y_pred - y_true) / denom))


def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="median")),
        ]
    )
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )
```

```python
    pre = ColumnTransformer(
        transformers=[
            ("num", num_pipe, NUM_COLS),
            ("cat", cat_pipe, CAT_COLS),
        ],
        remainder="drop",
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])


def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0, random_state=0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }


def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        return []

    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 10:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 3)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()

    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()

        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]

    return folds


def _score_fold(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    mae = float(np.mean(np.abs(y_true - y_pred)))
    wape = _wape(y_true, y_pred)
    smape = _smape(y_true, y_pred)
    return {"mae": mae, "wape": wape, "smape": smape}
```

```
def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(FEATURES_PATH).copy()

    need = {"fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "factor_tallos_dia_clipped"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"features_curva_cosecha_bloque_dia sin columnas: {sorted(miss)}")

    # target
    df["y"] = pd.to_numeric(df["factor_tallos_dia_clipped"], errors="coerce")

    # training rows: donde hay y finito
    train = df[np.isfinite(df["y"].to_numpy())].copy()
    if train.empty:
        raise ValueError("No hay filas con target (factor_tallos_dia_clipped) finito. Revisa real coverage.")

    # asegurar columnas
    for c in NUM_COLS:
        if c not in train.columns:
            train[c] = np.nan
    for c in CAT_COLS:
        if c not in train.columns:
            train[c] = "UNKNOWN"

    # folds temporales
    folds = _time_folds(train["fecha"], n_folds=4)
    if not folds:
        raise ValueError("No pude construir splits temporales. Revisa rango de fechas en FEATURES.")

    X_all = train[NUM_COLS + CAT_COLS]
    y_all = train["y"].to_numpy(dtype=float)

    models = _candidate_models()

    summary: dict = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "features_path": str(FEATURES_PATH),
        "n_rows_train_total": int(len(train)),
        "n_folds": int(len(folds)),
        "best_model": None,
        "metrics": {},
        "model_path": None,
    }

    best_name = None
    best_score = None
    best_model = None

    for name, model in models.items():
        pipe = _make_pipeline(model)

        fold_stats = []
        for tr_idx, va_idx in folds:
            X_tr = X_all.iloc[tr_idx]
            y_tr = y_all[tr_idx]
            X_va = X_all.iloc[va_idx]
            y_va = y_all[va_idx]

            pipe.fit(X_tr, y_tr)
            pred = pipe.predict(X_va)

            fold_stats.append(_score_fold(y_va, pred))

        maes = np.array([m["mae"] for m in fold_stats], dtype=float)
        wapes = np.array([m["wape"] for m in fold_stats], dtype=float)

        mae_mean = float(np.nanmean(maes))
        mae_std = float(np.nanstd(maes))
        wape_mean = float(np.nanmean(wapes))

        # score compuesto (más bajo es mejor)
        score = 0.60 * mae_mean + 0.30 * (wape_mean if np.isfinite(wape_mean) else mae_mean) + 0.10 * mae_std

        summary["metrics"][name] = {
            "mae_mean": mae_mean,
            "mae_std": mae_std,
            "wape_mean": wape_mean,
            "score": float(score),
            "n_rows": int(len(train)),
            "n_folds": int(len(folds)),
        }

        if (best_score is None) or (score < best_score):
            best_score = score
            best_name = name
            best_model = model

    assert best_name is not None and best_model is not None

    # fit final
    best_pipe = _make_pipeline(best_model)
    best_pipe.fit(X_all, y_all)

    model_path = out_dir / "model_curva_tallos_dia.joblib"
    dump(best_pipe, model_path)

    summary["best_model"] = best_name
    summary["model_path"] = str(model_path).replace("\\", "/")

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"[ML1 curva_tallos_dia] best={best_name} score={best_score:.6f}")
    print(f"OK -> {out_dir}/ (model + metrics.json)")


if __name__ == "__main__":
    main()

----------------------------------------------------------
[46/65] FILE: \src\models\ml1\train_desp_poscosecha_ml1.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

DIM_PATH = Path("data/silver/dim_mermas_ajuste_fecha_post_destino.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/desp_poscosecha")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino"]
```

```
def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )
    pre = ColumnTransformer(
        transformers=[("num", num_pipe, NUM_COLS), ("cat", cat_pipe, CAT_COLS)],
        remainder="drop",
        sparse_threshold=0.0,  # dense para HGB
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])


def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0, random_state=0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }


def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        return []
    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []
    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]
    folds = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        vs, ve = vb.min(), vb.max()
        tr = (all_dates < vs).to_numpy()
        va = ((all_dates >= vs) & (all_dates <= ve)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))
    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds


def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    dim = read_parquet(DIM_PATH).copy()
    dim.columns = [str(c).strip() for c in dim.columns]
```

```
need = {"fecha_post", "destino", "factor_desp"}
miss = need - set(dim.columns)
if miss:
    raise ValueError(f"dim_mermas_ajuste_fecha_post_destino sin columnas: {sorted(miss)}")

df = dim.copy()
df["fecha_post"] = pd.to_datetime(df["fecha_post"], errors="coerce").dt.normalize()
df["destino"] = df["destino"].astype(str).str.upper().str.strip()
df["y"] = pd.to_numeric(df["factor_desp"], errors="coerce")

df = df[df["fecha_post"].notna() & df["destino"].notna() & df["y"].notna()].copy()
df["y"] = df["y"].clip(lower=0.05, upper=1.00)

if len(df) < 100:
    raise ValueError(f"Poca data para entrenar desp (n={len(df)}).")

df["dow"] = df["fecha_post"].dt.dayofweek.astype("Int64")
df["month"] = df["fecha_post"].dt.month.astype("Int64")
df["weekofyear"] = df["fecha_post"].dt.isocalendar().week.astype("Int64")

X = df[NUM_COLS + CAT_COLS].copy()
y = df["y"].astype(float).to_numpy()

folds = _time_folds(df["fecha_post"], n_folds=4)
if not folds:
    raise ValueError("No pude armar folds temporales para desp.")

models = _candidate_models()

best_name, best_score, best_model = None, None, None
all_metrics = {}

for name, model in models.items():
    pipe = _make_pipeline(model)
    maes = []
    for tr_idx, va_idx in folds:
        pipe.fit(X.iloc[tr_idx], y[tr_idx])
        pred = pipe.predict(X.iloc[va_idx])
        maes.append(float(np.mean(np.abs(y[va_idx] - pred))))
    mae_mean = float(np.mean(maes))
    mae_std = float(np.std(maes))
    score = 0.85 * mae_mean + 0.15 * mae_std

    all_metrics[name] = {"mae_mean": mae_mean, "mae_std": mae_std, "score": float(score),
                         "n_rows": int(len(X)), "n_folds": int(len(folds))}
    if (best_score is None) or (score < best_score):
        best_score, best_name, best_model = score, name, model

best_pipe = _make_pipeline(best_model)
best_pipe.fit(X, y)

model_path = out_dir / "model.joblib"
dump(best_pipe, model_path)

summary = {
    "version": version,
    "created_at_utc": datetime.utcnow().isoformat(),
    "dim_path": str(DIM_PATH).replace("\\", "/"),
    "target": "factor_desp",
    "clip_range_apply": [0.05, 1.00],
    "features": {"num": NUM_COLS, "cat": CAT_COLS},
    "best_model": best_name,
    "metrics": all_metrics,
    "model_path": str(model_path).replace("\\", "/"),
}

with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(summary, f, ensure_ascii=False, indent=2)

print(f"OK -> {out_dir}/ (model.joblib + metrics.json) | best={best_name} score={best_score:.6f}")
```

```
if __name__ == "__main__":
    main()

--------------------------------------------------------
[47/65] FILE: \src\models\ml1\train_dh_poscosecha_ml1.py
--------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

FACT_PATH = Path("data/silver/fact_hidratacion_real_post_grado_destino.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/dh_poscosecha")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]  # OJO: aquí vamos a forzar ambos a string para evitar dtype mixto


# =============================================================================
# Utils
# =============================================================================
def _wape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = float(np.sum(np.abs(y_true)))
    if denom <= 1e-12:
        return float("nan")
    return float(np.sum(np.abs(y_true - y_pred)) / denom)


def _make_ohe() -> OneHotEncoder:
    # compat sklearn viejo/nuevo
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)


def _make_pipeline(model) -> Pipeline:
    # num -> median
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])

    # cat -> constant (evita error "most_frequent ... could not convert string to float")
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="constant", fill_value="UNKNOWN")),
            ("onehot", _make_ohe()),
        ]
    )

    pre = ColumnTransformer(
        transformers=[
            ("num", num_pipe, NUM_COLS),
            ("cat", cat_pipe, CAT_COLS),
        ],
        remainder="drop",
        sparse_threshold=0.0,  # fuerza salida densa
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])


def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0),  # Ridge NO lleva random_state
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }


def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        return []

    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds: list[tuple[np.ndarray, np.ndarray]] = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()

        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()

        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds


def _score_fold(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    mae = float(np.mean(np.abs(y_true - y_pred)))
    wape = _wape(y_true, y_pred)
    return {"mae": mae, "wape": wape}


# ==============================================================================
# Main
# ==============================================================================
def main() -> None:
    if not FACT_PATH.exists():
        raise FileNotFoundError(f"No existe: {FACT_PATH}")
```

```
version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
out_dir = REGISTRY_ROOT / version
out_dir.mkdir(parents=True, exist_ok=True)

fact = read_parquet(FACT_PATH).copy()
fact.columns = [str(c).strip() for c in fact.columns]

need = {"fecha_cosecha", "fecha_post", "dh_dias", "grado", "destino"}
miss = need - set(fact.columns)
if miss:
    raise ValueError(f"fact_hidratacion_real_post_grado_destino sin columnas: {sorted(miss)}")

df = fact.copy()

# Canon básico
df["fecha_cosecha"] = pd.to_datetime(df["fecha_cosecha"], errors="coerce").dt.normalize()
df["dh_dias"] = pd.to_numeric(df["dh_dias"], errors="coerce")
df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")

# destino: string robusto
df["destino"] = df["destino"].astype(str).str.upper().str.strip()

# filtros target
df = df[df["fecha_cosecha"].notna()].copy()
df = df[df["dh_dias"].notna()].copy()
df = df[df["dh_dias"].between(0, 30)].copy()
df = df[df["grado"].notna()].copy()

if df.empty:
    raise ValueError("No hay datos válidos para entrenar DH.")

# Features calendario (basadas en fecha_cosecha)
df["dow"] = df["fecha_cosecha"].dt.dayofweek
df["month"] = df["fecha_cosecha"].dt.month
df["weekofyear"] = df["fecha_cosecha"].dt.isocalendar().week.astype(int)

# Fuerza numérico limpio (por si vienen objetos raros)
for c in NUM_COLS:
    df[c] = pd.to_numeric(df[c], errors="coerce")

# >>> FIX CLAVE: fuerza cat a string (evita dtype mixto que dispara el error)
df["grado"] = df["grado"].astype("Int64").astype(str).replace("<NA>", "UNKNOWN")
df["destino"] = df["destino"].fillna("UNKNOWN").astype(str)

X = df[NUM_COLS + CAT_COLS].copy()
y = df["dh_dias"].astype(float).to_numpy()

folds = _time_folds(df["fecha_cosecha"], n_folds=4)
if not folds:
    raise ValueError("No pude armar folds temporales para DH.")

models = _candidate_models()

best_name: str | None = None
best_score: float | None = None
best_model = None
all_metrics: dict[str, dict] = {}

for name, model in models.items():
    pipe = _make_pipeline(model)
    fold_stats = []

    for tr_idx, va_idx in folds:
        X_tr = X.iloc[tr_idx]
        y_tr = y[tr_idx]
        X_va = X.iloc[va_idx]
        y_va = y[va_idx]

        pipe.fit(X_tr, y_tr)
        pred = pipe.predict(X_va)

        fold_stats.append(_score_fold(y_va, pred))
        maes = np.array([m["mae"] for m in fold_stats], dtype=float)
        wapes = np.array([m["wape"] for m in fold_stats], dtype=float)

        mae_mean = float(np.nanmean(maes))
        mae_std = float(np.nanstd(maes))
        wape_mean = float(np.nanmean(wapes))

        # score: prioriza MAE + WAPE, con penalización por varianza
        score = 0.75 * mae_mean + 0.15 * (wape_mean if np.isfinite(wape_mean) else mae_mean) + 0.10 * mae_std

        all_metrics[name] = {
            "mae_mean": mae_mean,
            "mae_std": mae_std,
            "wape_mean": wape_mean,
            "score": float(score),
            "n_rows": int(len(X)),
            "n_folds": int(len(folds)),
        }

        if (best_score is None) or (score < best_score):
            best_score = score
            best_name = name
            best_model = model

    assert best_name is not None and best_model is not None and best_score is not None

    best_pipe = _make_pipeline(best_model)
    best_pipe.fit(X, y)

    model_path = out_dir / "model.joblib"
    dump(best_pipe, model_path)

    summary = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "fact_path": str(FACT_PATH).replace("\\", "/"),
        "target": "dh_dias",
        "clip_range_apply": [0, 30],
        "features": {"num": NUM_COLS, "cat": CAT_COLS},
        "best_model": best_name,
        "metrics": all_metrics,
        "model_path": str(model_path).replace("\\", "/"),
    }

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {out_dir}/ (model.joblib + metrics.json) | best={best_name} score={best_score:.6f}")


if __name__ == "__main__":
    main()

--------------------------------------------------
[48/65] FILE: \src\models\ml1\train_dist_grado.py
--------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")


# -------------------------
# Config
# -------------------------
FEATURES_PATH = Path("data/features/features_cosecha_bloque_fecha.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/dist_grado")

# numéricas (si faltan, se crean con NaN)
NUM_COLS = [
    # etapa/progreso
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    # clima
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    # estado térmico SP
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    # calendario
    "dow",
    "month",
    "weekofyear",
    # baseline
    "share_grado_baseline",
]

# categóricas (si faltan, UNKNOWN)
CAT_COLS = [
    "variedad_canon",  # <-- clave
    "tipo_sp",
    "area",
]


# -------------------------
# Metrics
# -------------------------
def _wape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = np.sum(np.abs(y_true))
    if denom <= 1e-12:
        return float(np.nan)
    return float(np.sum(np.abs(y_true - y_pred)) / denom)


def _smape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = (np.abs(y_true) + np.abs(y_pred))
    denom = np.where(denom < 1e-12, 1e-12, denom)
    return float(np.mean(2.0 * np.abs(y_pred - y_true) / denom))


def _score_fold(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    mae = float(np.mean(np.abs(y_true - y_pred)))
    wape = _wape(y_true, y_pred)
    smape = _smape(y_true, y_pred)
    return {"mae": mae, "wape": wape, "smape": smape}


# -------------------------
# Pipeline / models
# -------------------------
def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="median")),
        ]
    )
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )

    pre = ColumnTransformer(
        transformers=[
            ("num", num_pipe, NUM_COLS),
            ("cat", cat_pipe, CAT_COLS),
        ],
        remainder="drop",
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])


def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0, random_state=0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=300,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }


# -------------------------
# Time folds
# -------------------------
def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    """
    Forward-chaining por fechas únicas.
    Fallback a holdout 80/20 si hay poca data.
    """
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        return []

    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()

        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]

    return folds


# -------------------------
# Main
# -------------------------
def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(FEATURES_PATH)

    # Compat: si viene "variedad" y no "variedad_canon", lo usamos.
    if "variedad_canon" not in df.columns and "variedad" in df.columns:
        df["variedad_canon"] = df["variedad"]

    # Validaciones mínimas
    need = {"fecha", "bloque_base", "grado", "share_grado_real", "share_grado_baseline"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"features_cosecha_bloque_fecha.parquet sin columnas: {sorted(miss)}")

    # Training dataset: solo donde hay target real
    train_df = df[df["share_grado_real"].notna()].copy()

    # (Opcional) entrenar solo en ventana real si existe
    if "en_ventana_cosecha_real" in train_df.columns:
        train_df = train_df[train_df["en_ventana_cosecha_real"] == 1].copy()

    # Asegurar columnas
    for c in NUM_COLS:
        if c not in train_df.columns:
            train_df[c] = np.nan
    for c in CAT_COLS:
        if c not in train_df.columns:
            train_df[c] = "UNKNOWN"

    # Tipos
    train_df["grado"] = pd.to_numeric(train_df["grado"], errors="coerce").astype("Int64")

    grades = sorted(train_df["grado"].dropna().unique().tolist())
    if not grades:
        raise ValueError("No encontré grados para entrenar (columna 'grado').")

    models = _candidate_models()

    summary: dict = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "features_path": str(FEATURES_PATH),
        "n_rows_train_total": int(len(train_df)),
        "n_folds_default": 4,
        "grades": grades,
        "best_by_grade": {},
    }

    # Entrenar por grado
    for g in grades:
        gdf = train_df[train_df["grado"] == g].copy()
        # seguridad extra: target numérico
        y = pd.to_numeric(gdf["share_grado_real"], errors="coerce").to_numpy(dtype=float)
        X = gdf[NUM_COLS + CAT_COLS].copy()

        mask = np.isfinite(y)
        X = X.loc[mask].reset_index(drop=True)
        y = y[mask]

        if len(y) < 30:
            # muy poca data: igual entrenamos, pero con holdout mínimo
            pass

        g_folds = _time_folds(gdf.loc[mask, "fecha"], n_folds=4)
        if not g_folds:
            d = pd.to_datetime(gdf.loc[mask, "fecha"], errors="coerce").dt.normalize()
            cut = d.quantile(0.8)
            tr_idx = np.where(d < cut)[0]
            va_idx = np.where(d >= cut)[0]
            if len(tr_idx) == 0 or len(va_idx) == 0:
                # no hay split, entrenamos sin validación (pero dejamos métricas NaN)
                g_folds = []
            else:
                g_folds = [(tr_idx, va_idx)]

        best_name = None
        best_score = None
        best_model = None
        all_metrics = {}

        for name, model in models.items():
            pipe = _make_pipeline(model)

            fold_stats = []
            if g_folds:
                for tr_idx, va_idx in g_folds:
                    X_tr = X.iloc[tr_idx]
                    y_tr = y[tr_idx]
                    X_va = X.iloc[va_idx]
                    y_va = y[va_idx]

                    pipe.fit(X_tr, y_tr)
                    pred = pipe.predict(X_va)

                    fold_stats.append(_score_fold(y_va, pred))

                maes = np.array([m["mae"] for m in fold_stats], dtype=float)
                wapes = np.array([m["wape"] for m in fold_stats], dtype=float)
                smapes = np.array([m["smape"] for m in fold_stats], dtype=float)

                mae_mean = float(np.nanmean(maes))
                mae_std = float(np.nanstd(maes))
                wape_mean = float(np.nanmean(wapes))
                smape_mean = float(np.nanmean(smapes))

                # score compuesto (más bajo es mejor)
                score = 0.55 * mae_mean + 0.30 * (wape_mean if np.isfinite(wape_mean) else mae_mean) + 0.15 *
mae_std
            else:
                # sin folds posibles
                mae_mean = float("nan")
                mae_std = float("nan")
                wape_mean = float("nan")
                smape_mean = float("nan")
                score = float("inf")

            all_metrics[name] = {
                "mae_mean": mae_mean,
                "mae_std": mae_std,
                "wape_mean": wape_mean,
                "smape_mean": smape_mean,
                "score": float(score),
                "n_rows": int(len(X)),
                "n_folds": int(len(g_folds)),
            }

            if (best_score is None) or (score < best_score):
                best_score = score
                best_name = name
                best_model = model

        # Fit final con todo el historial (grado g)
        assert best_name is not None and best_model is not None
        best_pipe = _make_pipeline(best_model)
        best_pipe.fit(X, y)

        # Guardar artefacto
        model_path = out_dir / f"model_grade_{str(int(g))}.joblib"
        dump(best_pipe, model_path)

        summary["best_by_grade"][str(int(g))] = {
            "best_model": best_name,
            "metrics": all_metrics,
            "model_path": str(model_path).replace("\\", "/"),
        }

        print(f"[ML1 dist_grado] grado={int(g)} best={best_name} score={best_score}")

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {out_dir}/ (models + metrics.json)")


if __name__ == "__main__":
    main()

--------------------------------------------------------
[49/65] FILE: \src\models\ml1\train_harvest_window_ml1.py
--------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")


FEATURES_PATH = Path("data/features/features_harvest_window_ml1.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/harvest_window")


NUM_COLS = [
    "tallos_proy",
    "sp_month",
    "sp_weekofyear",
    "sp_doy",
    "sp_dow",
]

CAT_COLS = [
    "variedad_canon",
    "area",
    "tipo_sp",
]


def _make_pipe() -> Pipeline:
    num_pipe = Pipeline([("imputer", SimpleImputer(strategy="median"))])
    cat_pipe = Pipeline(
        [
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )
    pre = ColumnTransformer(
        [
            ("num", num_pipe, NUM_COLS),
            ("cat", cat_pipe, CAT_COLS),
        ],
        remainder="drop",
    )
    # HGB: robusto, no se va al infinito, generaliza bien con onehot sparse
    model = HistGradientBoostingRegressor(
        loss="squared_error",
        random_state=0,
        max_depth=6,
        learning_rate=0.07,
        max_iter=500,
        min_samples_leaf=25,  # pooling implícito para segmentos chicos
        l2_regularization=1.0,
        early_stopping=True,
        validation_fraction=0.15,
    )
    return Pipeline([("pre", pre), ("model", model)])


def _time_split(df: pd.DataFrame, date_col: str = "fecha_sp") -> tuple[np.ndarray, np.ndarray]:
    d = pd.to_datetime(df[date_col], errors="coerce").dt.normalize()
    ok = d.notna()
    if ok.sum() < 50:
        # fallback simple
        idx = np.arange(len(df))
        cut = int(len(idx) * 0.8)
        return idx[:cut], idx[cut:]

    uniq = np.array(sorted(d[ok].unique()))
    cut = uniq[int(len(uniq) * 0.8)]
    tr = np.where(d < cut)[0]
    va = np.where(d >= cut)[0]
    if len(tr) == 0 or len(va) == 0:
        idx = np.arange(len(df))
        cut2 = int(len(idx) * 0.8)
        return idx[:cut2], idx[cut2:]
    return tr, va


def _mae(y, p) -> float:
    y = np.asarray(y, dtype=float)
    p = np.asarray(p, dtype=float)
    return float(np.nanmean(np.abs(y - p)))


def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(FEATURES_PATH).copy()
    need = {"ciclo_id", "fecha_sp", "variedad_canon", "area", "tipo_sp"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"features_harvest_window_ml1 sin columnas: {sorted(miss)}")
    # asegurar columnas
    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    # ---- train start offset
    df_start = df[df["d_start_real"].notna()].copy()
    if df_start.empty:
        raise ValueError("No hay d_start_real para entrenar.")

    Xs = df_start[NUM_COLS + CAT_COLS]
    ys = pd.to_numeric(df_start["d_start_real"], errors="coerce").to_numpy(dtype=float)

    tr, va = _time_split(df_start, "fecha_sp")
    pipe_start = _make_pipe()
    pipe_start.fit(Xs.iloc[tr], ys[tr])
    pred_va = pipe_start.predict(Xs.iloc[va])
    mae_start = _mae(ys[va], pred_va)

    # ---- train harvest days
    df_days = df[df["n_harvest_days_real"].notna()].copy()
    if df_days.empty:
        raise ValueError("No hay n_harvest_days_real para entrenar.")

    Xd = df_days[NUM_COLS + CAT_COLS]
    yd = pd.to_numeric(df_days["n_harvest_days_real"], errors="coerce").to_numpy(dtype=float)

    tr2, va2 = _time_split(df_days, "fecha_sp")
    pipe_days = _make_pipe()
    pipe_days.fit(Xd.iloc[tr2], yd[tr2])
    pred_va2 = pipe_days.predict(Xd.iloc[va2])
    mae_days = _mae(yd[va2], pred_va2)

    # fit final full
    pipe_start.fit(Xs, ys)
    pipe_days.fit(Xd, yd)

    dump(pipe_start, out_dir / "model_start_offset.joblib")
    dump(pipe_days, out_dir / "model_harvest_days.joblib")

    meta = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "features_path": str(FEATURES_PATH).replace("\\", "/"),
        "n_train_start": int(len(df_start)),
        "n_train_days": int(len(df_days)),
        "mae_start_days_holdout": mae_start,
        "mae_harvest_days_holdout": mae_days,
        "num_cols": NUM_COLS,
        "cat_cols": CAT_COLS,
        "clips": {"d_start": [0, 180], "n_harvest_days": [1, 180]},
    }
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(meta, f, ensure_ascii=False, indent=2)

    print(f"[ML1 harvest_window] version={version}")
    print(f"MAE start_offset (days): {mae_start:.3f}")
    print(f"MAE harvest_days (days): {mae_days:.3f}")
    print(f"OK -> {out_dir}/ (models + metrics.json)")


if __name__ == "__main__":
    main()

----------------------------------------------------------
[50/65] FILE: \src\models\ml1\train_hidr_poscosecha_ml1.py
----------------------------------------------------------
from __future__ import annotations
```

```python
from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

FACT_PATH = Path("data/silver/fact_hidratacion_real_post_grado_destino.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/hidr_poscosecha")

# numéricas (calendario)
NUM_COLS = ["dow", "month", "weekofyear"]

# categóricas separadas por tipo (EVITA el crash de most_frequent con mezcla dtype)
CAT_STR_COLS = ["destino"]   # string
CAT_NUM_COLS = ["grado"]     # num/cat


def _make_ohe() -> OneHotEncoder:
    # compat sklearn viejo/nuevo
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)


def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])

    cat_str_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="constant", fill_value="UNKNOWN")),
            ("onehot", _make_ohe()),
        ]
    )

    cat_num_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="constant", fill_value=-1)),
            ("onehot", _make_ohe()),
        ]
    )

    pre = ColumnTransformer(
        transformers=[
            ("num", num_pipe, NUM_COLS),
            ("cat_str", cat_str_pipe, CAT_STR_COLS),
            ("cat_num", cat_num_pipe, CAT_NUM_COLS),
        ],
        remainder="drop",
        sparse_threshold=0.0,  # fuerza denso (evita temas con HGB)
    )

    return Pipeline(steps=[("pre", pre), ("model", model)])


def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }


def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    if d.isna().all():
        return []

    uniq = np.array(sorted(d.dropna().unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds: list[tuple[np.ndarray, np.ndarray]] = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()
        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds


def main() -> None:
    if not FACT_PATH.exists():
        raise FileNotFoundError(f"No existe: {FACT_PATH}")

    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    fact = read_parquet(FACT_PATH).copy()
    fact.columns = [str(c).strip() for c in fact.columns]

    need = {"fecha_cosecha", "grado", "destino"}
    miss = need - set(fact.columns)
    if miss:
        raise ValueError(f"fact_hidratacion_real_post_grado_destino sin columnas: {sorted(miss)}")

    # target: factor_hidr si existe; si no, 1+hidr_pct
    if "factor_hidr" in fact.columns:
        fact["y"] = pd.to_numeric(fact["factor_hidr"], errors="coerce")
        target_name = "factor_hidr"
    elif "hidr_pct" in fact.columns:
        fact["y"] = 1.0 + pd.to_numeric(fact["hidr_pct"], errors="coerce")
        target_name = "1+hidr_pct"
    else:
        raise ValueError("No encuentro ni factor_hidr ni hidr_pct en fact_hidratacion_real_post_grado_destino.
")

    fact["fecha_cosecha"] = pd.to_datetime(fact["fecha_cosecha"], errors="coerce").dt.normalize()

    # grado num/cat (se queda num, se onehotea como categoría)
    fact["grado"] = pd.to_numeric(fact["grado"], errors="coerce")

    # destino str
    fact["destino"] = fact["destino"].astype(str).str.upper().str.strip()
    fact.loc[fact["destino"].isin(["", "NAN", "NONE", "NULL"]), "destino"] = np.nan

    # peso base para ponderación si existe
    w = None
    if "peso_base_g" in fact.columns:
        w = pd.to_numeric(fact["peso_base_g"], errors="coerce")
    elif "tallos" in fact.columns:
        w = pd.to_numeric(fact["tallos"], errors="coerce")

    df = fact[fact["fecha_cosecha"].notna() & fact["grado"].notna() & fact["y"].notna()].copy()

    # caps razonables (alineado a tu lógica histórica)
    df["y"] = pd.to_numeric(df["y"], errors="coerce").clip(lower=0.80, upper=3.00)

    if len(df) < 200:
        raise ValueError(f"Poca data para entrenar hidr (n={len(df)}).")

    # features calendario
    df["dow"] = df["fecha_cosecha"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha_cosecha"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha_cosecha"].dt.isocalendar().week.astype("Int64")

    X = df[NUM_COLS + CAT_STR_COLS + CAT_NUM_COLS].copy()
    y = df["y"].astype(float).to_numpy()

    # sample_weight (opcional)
    sample_weight = None
    if w is not None:
        ww = w.loc[df.index]
        ww = pd.to_numeric(ww, errors="coerce").fillna(0.0).astype(float)
        # si viene todo 0, no sirve
        if float(ww.sum()) > 0:
            sample_weight = ww.to_numpy()

    folds = _time_folds(df["fecha_cosecha"], n_folds=4)
    if not folds:
        raise ValueError("No pude armar folds temporales para hidr.")

    models = _candidate_models()

    best_name = None
    best_score = None
    best_model = None
    all_metrics: dict[str, dict] = {}

    for name, model in models.items():
        pipe = _make_pipeline(model)
        maes = []
        for tr_idx, va_idx in folds:
            X_tr, y_tr = X.iloc[tr_idx], y[tr_idx]
            X_va, y_va = X.iloc[va_idx], y[va_idx]

            fit_kwargs = {}
            if sample_weight is not None:
                fit_kwargs["model__sample_weight"] = sample_weight[tr_idx]

            pipe.fit(X_tr, y_tr, **fit_kwargs)
            pred = pipe.predict(X_va)
            maes.append(float(np.mean(np.abs(y_va - pred))))

        mae_mean = float(np.mean(maes))
        mae_std = float(np.std(maes))
        score = 0.85 * mae_mean + 0.15 * mae_std

        all_metrics[name] = {
            "mae_mean": mae_mean,
            "mae_std": mae_std,
            "score": float(score),
            "n_rows": int(len(X)),
            "n_folds": int(len(folds)),
            "uses_sample_weight": bool(sample_weight is not None),
        }

        if (best_score is None) or (score < best_score):
            best_score = score
            best_name = name
            best_model = model

    assert best_name is not None and best_model is not None

    best_pipe = _make_pipeline(best_model)

    fit_kwargs = {}
    if sample_weight is not None:
        fit_kwargs["model__sample_weight"] = sample_weight

    best_pipe.fit(X, y, **fit_kwargs)

    model_path = out_dir / "model.joblib"
    dump(best_pipe, model_path)

    summary = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "fact_path": str(FACT_PATH).replace("\\", "/"),
        "target": target_name,
        "clip_range_apply": [0.80, 3.00],
        "features": {
            "num": NUM_COLS,
            "cat_str": CAT_STR_COLS,
            "cat_num": CAT_NUM_COLS,
        },
        "best_model": best_name,
        "metrics": all_metrics,
        "uses_sample_weight": bool(sample_weight is not None),
        "model_path": str(model_path).replace("\\", "/"),
    }

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {out_dir}/ (model.joblib + metrics.json) | best={best_name} score={best_score:.6f}")


if __name__ == "__main__":
    main()

-------------------------------------------------------
[51/65] FILE: \src\models\ml1\train_peso_tallo_grado.py
-------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
```

```python
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor
from sklearn.dummy import DummyRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

FEATURES_PATH = Path("data/features/features_peso_tallo_grado_bloque_dia.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/peso_tallo_grado")

# =========================
# Config
# =========================
CLIP_LOW, CLIP_HIGH = 0.60, 1.60

NUM_COLS = [
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
    "peso_tallo_baseline_g",
]

CAT_COLS = ["variedad_canon", "tipo_sp", "area"]

# Candidatos para reconstruir target si viene vacío
REAL_W_CANDS = [
    "peso_tallo_real_g",
    "peso_tallo_real",
    "peso_tallo_g_real",
    "peso_tallo_obs_g",
    "peso_tallo_prom_g",
    "peso_tallo_avg_g",
    "peso_tallo_mediana_g",
    "peso_tallo_g",
    "peso_real_g",
    "peso_real",
]
BASE_W_CANDS = [
    "peso_tallo_baseline_g",
    "peso_tallo_baseline",
    "peso_baseline_g",
    "peso_base_g",
]


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()
```

```python
def _wape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = float(np.sum(np.abs(y_true)))
    if denom <= 1e-12:
        return float("nan")
    return float(np.sum(np.abs(y_true - y_pred)) / denom)


def _smape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = (np.abs(y_true) + np.abs(y_pred))
    denom = np.where(denom < 1e-12, 1e-12, denom)
    return float(np.mean(2.0 * np.abs(y_pred - y_true) / denom))


def _make_ohe() -> OneHotEncoder:
    # compat sklearn viejo/nuevo
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)


def _make_pipeline(model) -> Pipeline:
    # num -> median (robusto)
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])

    # cat -> constant (evita bugs dtype mixto)
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="constant", fill_value="UNKNOWN")),
            ("onehot", _make_ohe()),
        ]
    )

    pre = ColumnTransformer(
        transformers=[
            ("num", num_pipe, NUM_COLS),
            ("cat", cat_pipe, CAT_COLS),
        ],
        remainder="drop",
        sparse_threshold=0.0,  # fuerza dense
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])


def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=300,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }


def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    if d.isna().all():
        return []

    uniq = np.array(sorted(d.dropna().unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds: list[tuple[np.ndarray, np.ndarray]] = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()
        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds


def _score_fold(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    mae = float(np.mean(np.abs(y_true - y_pred)))
    wape = _wape(y_true, y_pred)
    smape = _smape(y_true, y_pred)
    return {"mae": mae, "wape": wape, "smape": smape}


def _pick_col(df: pd.DataFrame, cands: list[str]) -> str | None:
    cols = list(df.columns)
    for c in cands:
        if c in cols:
            return c
    # match case-insensitive / trimmed
    norm = {str(c).strip().upper(): c for c in cols}
    for c in cands:
        k = str(c).strip().upper()
        if k in norm:
            return norm[k]
    return None


def _ensure_target(df: pd.DataFrame) -> tuple[pd.DataFrame, dict]:
    """
    Garantiza factor_peso_tallo_clipped:
    - si ya hay valores -> listo
    - si está vacío -> intenta reconstruir desde peso_real / peso_baseline
    """
    info = {}

    if "factor_peso_tallo_clipped" in df.columns:
        nn = int(df["factor_peso_tallo_clipped"].notna().sum())
        info["target_present_nonnull"] = nn
        if nn > 0:
            return df, info

    real_col = _pick_col(df, REAL_W_CANDS)
    base_col = _pick_col(df, BASE_W_CANDS)

    info["real_col_used"] = real_col
    info["base_col_used"] = base_col

    if real_col is None or base_col is None:
        # no se puede reconstruir
        if "factor_peso_tallo_clipped" not in df.columns:
            df["factor_peso_tallo_clipped"] = np.nan
        info["target_rebuilt"] = False
        info["target_rebuilt_reason"] = "missing real/base columns"
        return df, info
```

```python
    real = pd.to_numeric(df[real_col], errors="coerce")
    base = pd.to_numeric(df[base_col], errors="coerce")

    factor = np.where((base > 0) & np.isfinite(base), real / base, np.nan)
    factor = pd.Series(factor, index=df.index)
    factor = factor.replace([np.inf, -np.inf], np.nan)

    df["factor_peso_tallo_raw"] = factor
    df["factor_peso_tallo_clipped"] = pd.to_numeric(df["factor_peso_tallo_raw"], errors="coerce").clip(
        lower=CLIP_LOW, upper=CLIP_HIGH
    )

    info["target_rebuilt"] = True
    info["target_rebuilt_nonnull"] = int(df["factor_peso_tallo_clipped"].notna().sum())
    return df, info


def _fit_constant_pipeline(constant_value: float) -> Pipeline:
    """
    Modelo fallback: predice constante (ej. 1.0).
    Se "fitea" sobre 1 fila dummy para dejar el pipeline serializable.
    """
    model = DummyRegressor(strategy="constant", constant=float(constant_value))
    pipe = _make_pipeline(model)

    X_dummy = pd.DataFrame(
        {
            **{c: [0.0] for c in NUM_COLS},
            **{c: ["UNKNOWN"] for c in CAT_COLS},
        }
    )
    y_dummy = np.array([float(constant_value)], dtype=float)
    pipe.fit(X_dummy, y_dummy)
    return pipe


def main() -> None:
    if not FEATURES_PATH.exists():
        raise FileNotFoundError(f"No existe: {FEATURES_PATH}")

    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(FEATURES_PATH).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "peso_tallo_baseline_g"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"features_peso_tallo_grado_bloque_dia.parquet sin columnas: {sorted(miss)}")

    # Canon llaves/fechas
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_int(df["bloque_base"])
    df["grado"] = _canon_int(df["grado"])
    df["variedad_canon"] = _canon_str(df["variedad_canon"])

    # Asegurar cat cols
    for c in ["tipo_sp", "area"]:
        if c not in df.columns:
            df[c] = "UNKNOWN"
    df["tipo_sp"] = _canon_str(df["tipo_sp"].fillna("UNKNOWN"))
    df["area"] = _canon_str(df["area"].fillna("UNKNOWN"))

    # Asegurar num cols (evita strings raros en num)
    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
        df[c] = pd.to_numeric(df[c], errors="coerce")

    # Construir/garantizar target
```

```
df, tinfo = _ensure_target(df)

# Train set = donde target válido + fecha válida + grado válido
train_df = df[
    df["factor_peso_tallo_clipped"].notna()
    & df["fecha"].notna()
    & df["grado"].notna()
].copy()

grades = sorted(df["grado"].dropna().astype(int).unique().tolist())
models = _candidate_models()

summary: dict = {
    "version": version,
    "created_at_utc": datetime.utcnow().isoformat(),
    "features_path": str(FEATURES_PATH).replace("\\", "/"),
    "target": "factor_peso_tallo_clipped",
    "clip_range": [CLIP_LOW, CLIP_HIGH],
    "n_rows_total": int(len(df)),
    "n_rows_train_total": int(len(train_df)),
    "grades_seen": [int(g) for g in grades],
    "target_build_info": tinfo,
    "best_by_grade": {},
}

if not grades:
    # no grados => igual guardo métricas y salgo
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)
    print(f"[WARN] No hay grados en features. OK -> {out_dir}/metrics.json")
    return

if train_df.empty:
    # no hay target: guardo modelos constantes por grado para no romper downstream
    print("[WARN] No hay filas con target para entrenar. Se generan modelos constantes (1.0) por grado.")
    for g in grades:
        g = int(g)
        pipe = _fit_constant_pipeline(1.0)
        model_path = out_dir / f"model_grade_{g}.joblib"
        dump(pipe, model_path)
        summary["best_by_grade"][str(g)] = {
            "best_model": "dummy_const_1.0",
            "metrics": {},
            "model_path": str(model_path).replace("\\", "/"),
            "n_rows_grade": 0,
        }

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {out_dir}/ (models const + metrics.json)")
    return

# Entrenamiento por grado
for g in grades:
    g = int(g)
    gdf = train_df[train_df["grado"].astype(int) == g].copy()

    if len(gdf) < 30:
        # fallback modelo constante por grado
        pipe = _fit_constant_pipeline(1.0)
        model_path = out_dir / f"model_grade_{g}.joblib"
        dump(pipe, model_path)
        summary["best_by_grade"][str(g)] = {
            "best_model": "dummy_const_1.0",
            "metrics": {},
            "model_path": str(model_path).replace("\\", "/"),
            "n_rows_grade": int(len(gdf)),
            "note": "poca data; fallback const",
        }
        print(f"[ML1 peso_tallo] grado={g} -> fallback const=1.0 (poca data n={len(gdf)})")
        continue
```

```
# Features/target
X = gdf[NUM_COLS + CAT_COLS].copy()
for c in CAT_COLS:
    X[c] = _canon_str(X[c].fillna("UNKNOWN"))

y = pd.to_numeric(gdf["factor_peso_tallo_clipped"], errors="coerce").to_numpy(dtype=float)
mask = np.isfinite(y)
if mask.sum() < 30:
    pipe = _fit_constant_pipeline(1.0)
    model_path = out_dir / f"model_grade_{g}.joblib"
    dump(pipe, model_path)
    summary["best_by_grade"][str(g)] = {
        "best_model": "dummy_const_1.0",
        "metrics": {},
        "model_path": str(model_path).replace("\\", "/"),
        "n_rows_grade": int(len(gdf)),
        "note": "target inválido; fallback const",
    }
    print(f"[ML1 peso_tallo] grado={g} -> fallback const=1.0 (target inválido)")
    continue

X = X.loc[gdf.index[mask]].copy()
y = y[mask]

folds = _time_folds(gdf.loc[X.index, "fecha"], n_folds=4)
if not folds:
    # fallback split 80/20 por fecha
    d = pd.to_datetime(gdf.loc[X.index, "fecha"], errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        folds = []
    else:
        cut = d.quantile(0.8)
        d_arr = pd.to_datetime(gdf.loc[X.index, "fecha"], errors="coerce").dt.normalize().to_numpy()
        tr_idx = np.where(d_arr < np.datetime64(cut))[0]
        va_idx = np.where(d_arr >= np.datetime64(cut))[0]
        if len(tr_idx) > 0 and len(va_idx) > 0:
            folds = [(tr_idx, va_idx)]

if not folds:
    pipe = _fit_constant_pipeline(1.0)
    model_path = out_dir / f"model_grade_{g}.joblib"
    dump(pipe, model_path)
    summary["best_by_grade"][str(g)] = {
        "best_model": "dummy_const_1.0",
        "metrics": {},
        "model_path": str(model_path).replace("\\", "/"),
        "n_rows_grade": int(len(X)),
        "note": "no folds; fallback const",
    }
    print(f"[ML1 peso_tallo] grado={g} -> fallback const=1.0 (no folds)")
    continue

best_name = None
best_score = None
best_model = None
all_metrics: dict[str, dict] = {}

for name, model in models.items():
    pipe = _make_pipeline(model)
    fold_stats = []
    for tr_idx, va_idx in folds:
        X_tr = X.iloc[tr_idx]
        y_tr = y[tr_idx]
        X_va = X.iloc[va_idx]
        y_va = y[va_idx]

        pipe.fit(X_tr, y_tr)
        pred = pipe.predict(X_va)

        fold_stats.append(_score_fold(y_va, pred))
```

```
            maes = np.array([m["mae"] for m in fold_stats], dtype=float)
            wapes = np.array([m["wape"] for m in fold_stats], dtype=float)
            smapes = np.array([m["smape"] for m in fold_stats], dtype=float)

            mae_mean = float(np.nanmean(maes))
            mae_std = float(np.nanstd(maes))
            wape_mean = float(np.nanmean(wapes))
            smape_mean = float(np.nanmean(smapes))

            # score híbrido (MAE + estabilidad)
            score = 0.60 * mae_mean + 0.25 * (wape_mean if np.isfinite(wape_mean) else mae_mean) + 0.15 * mae_
std

            all_metrics[name] = {
                "mae_mean": mae_mean,
                "mae_std": mae_std,
                "wape_mean": wape_mean,
                "smape_mean": smape_mean,
                "score": float(score),
                "n_rows": int(len(X)),
                "n_folds": int(len(folds)),
            }

            if (best_score is None) or (score < best_score):
                best_score = score
                best_name = name
                best_model = model

        assert best_name is not None and best_model is not None

        best_pipe = _make_pipeline(best_model)
        best_pipe.fit(X, y)

        model_path = out_dir / f"model_grade_{g}.joblib"
        dump(best_pipe, model_path)

        summary["best_by_grade"][str(g)] = {
            "best_model": best_name,
            "metrics": all_metrics,
            "model_path": str(model_path).replace("\\", "/"),
            "n_rows_grade": int(len(X)),
        }

        print(f"[ML1 peso_tallo] grado={g} best={best_name} score={best_score:.6f}")

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {out_dir}/ (models + metrics.json)")


if __name__ == "__main__":
    main()

------------------------------------------------------------
[52/65] FILE: \src\models\ml2\apply_ajuste_poscosecha_ml2.py
------------------------------------------------------------
from __future__ import annotations

import argparse
import json
from pathlib import Path

import numpy as np
import pandas as pd
from joblib import load

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS_DIR = DATA / "models" / "ml2"

IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"

OUT_FINAL = GOLD / "pred_poscosecha_ml2_ajuste_grado_dia_bloque_destino_final.parquet"
OUT_BACKTEST = EVAL / "backtest_factor_ml2_ajuste_poscosecha.parquet"


# ----------------------------
# TZ-safe helpers
# ----------------------------
def _to_naive_utc(ts: pd.Timestamp) -> pd.Timestamp:
    if ts.tzinfo is None:
        return ts
    return ts.tz_convert("UTC").tz_localize(None)


def _as_of_date_naive() -> pd.Timestamp:
    # regla: hoy-1
    t = _to_naive_utc(pd.Timestamp.utcnow())
    return (t.normalize() - pd.Timedelta(days=1))


def _to_date_naive(s: pd.Series) -> pd.Series:
    dt = pd.to_datetime(s, errors="coerce")
    try:
        if getattr(dt.dt, "tz", None) is not None:
            dt = dt.dt.tz_convert("UTC").dt.tz_localize(None)
    except Exception:
        pass
    return dt.dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:
        if c in df.columns:
            return c
    raise KeyError("No encuentro columna fecha_post_pred (final/used/ml1/seed).")


def _resolve_ajuste_ml1(df: pd.DataFrame) -> str:
    for c in ["ajuste_ml1", "factor_ajuste_ml1", "factor_ajuste_seed", "factor_ajuste"]:
        if c in df.columns:
            return c
    raise KeyError("No encuentro ajuste ML1 (esperaba ajuste_ml1 o factor_ajuste_*).")


def _weight_series(df: pd.DataFrame) -> pd.Series:
    for c in ["tallos_w", "tallos", "tallos_total_ml2", "tallos_total"]:
        if c in df.columns:
            return pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    return pd.Series(1.0, index=df.index, dtype="float64")


def _latest_model(prefix: str = "ajuste_poscosecha_ml2_") -> Path:
    if not MODELS_DIR.exists():
        raise FileNotFoundError(f"No existe {MODELS_DIR}")
    files = sorted(MODELS_DIR.glob(f"{prefix}*.pkl"))
    if not files:
        raise FileNotFoundError(f"No encontré modelos en {MODELS_DIR} con prefijo {prefix}")
    return files[-1]
```

```
def _meta_path_from_model(model_path: Path) -> Path:
    # ? NO usar with_suffix("_meta.json") (inválido)
    p = str(model_path)
    if p.endswith(".pkl"):
        return Path(p.replace(".pkl", "_meta.json"))
    return model_path.with_suffix(".json")


def main(mode: str = "backtest", model_file: str | None = None) -> None:
    as_of = _as_of_date_naive()

    model_path = Path(model_file) if model_file else _latest_model()
    meta_path = _meta_path_from_model(model_path)

    meta = {}
    if meta_path.exists():
        with open(meta_path, "r", encoding="utf-8") as f:
            meta = json.load(f)

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    if "fecha" not in df.columns or "destino" not in df.columns:
        raise ValueError("Universe debe tener fecha y destino.")

    # ? tz-safe
    df["fecha"] = _to_date_naive(df["fecha"])
    df["destino"] = _canon_str(df["destino"])

    fecha_post_col = _resolve_fecha_post_pred(df)
    df[fecha_post_col] = _to_date_naive(df[fecha_post_col])

    ajuste_ml1_col = _resolve_ajuste_ml1(df)
    df[ajuste_ml1_col] = pd.to_numeric(df[ajuste_ml1_col], errors="coerce")

    # ? filtro as_of consistente
    df = df[df["fecha"].notna() & (df["fecha"] <= as_of)].copy()

    # features calendario
    df["dow"] = df[fecha_post_col].dt.dayofweek.astype("Int64")
    df["month"] = df[fecha_post_col].dt.month.astype("Int64")
    df["weekofyear"] = df[fecha_post_col].dt.isocalendar().week.astype("Int64")
    df["w"] = _weight_series(df)

    # columnas para el modelo
    NUM_COLS = ["dow", "month", "weekofyear"]
    CAT_COLS = ["destino"]
    if "grado" in df.columns:
        CAT_COLS.append("grado")
        df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")

    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    X = df[NUM_COLS + CAT_COLS]

    model = load(model_path)
    pred = model.predict(X)

    # delta: log(real/ml1)
    clip = float(meta.get("clip_delta", 1.2))
    df["delta_log_ajuste_ml2_raw"] = pd.to_numeric(pd.Series(pred), errors="coerce")
    df["delta_log_ajuste_ml2"] = df["delta_log_ajuste_ml2_raw"].clip(lower=-clip, upper=clip)

    # aplicar: ajuste_final = ajuste_ml1 * exp(delta)
    base = pd.to_numeric(df[ajuste_ml1_col], errors="coerce").replace(0, np.nan)
    df["factor_ajuste_final"] = (base * np.exp(df["delta_log_ajuste_ml2"])).astype(float)

    # guardarraíles
    clip_factor = meta.get("clip_factor_apply", [0.50, 2.00])
    try:
        lo, hi = float(clip_factor[0]), float(clip_factor[1])
    except Exception:
        lo, hi = 0.50, 2.00
    df["factor_ajuste_final"] = df["factor_ajuste_final"].clip(lower=lo, upper=hi)

    df["ml2_ajuste_model_file"] = model_path.name
    df["ml2_ajuste_clip_delta"] = clip
    df["as_of_date"] = as_of
    df["created_at"] = pd.Timestamp.utcnow()

    # backtest factor output + final output
    if mode.lower() == "backtest":
        bt_cols = ["fecha", fecha_post_col, "destino", "delta_log_ajuste_ml2", "w", "as_of_date", "created_at"
]
        if "grado" in df.columns:
            bt_cols.insert(3, "grado")
        bt = df[bt_cols].copy()
        EVAL.mkdir(parents=True, exist_ok=True)
        write_parquet(bt, OUT_BACKTEST)
        print(f"[OK] BACKTEST factor: {OUT_BACKTEST} rows={len(bt):,}")

    write_parquet(df, OUT_FINAL)
    print(f"[OK] BACKTEST final : {OUT_FINAL} rows={len(df):,}")
    print(f"     model={model_path.name} clip_delta=±{clip} as_of_date={as_of.date()}")


if __name__ == "__main__":
    p = argparse.ArgumentParser()
    p.add_argument("--mode", default="backtest", choices=["backtest", "prod"])
    p.add_argument("--model_file", default=None)
    args = p.parse_args()
    main(mode=args.mode, model_file=args.model_file)

----------------------------------------------------------
[53/65] FILE: \src\models\ml2\apply_desp_poscosecha_ml2.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json

import numpy as np
import pandas as pd
from joblib import load

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"

OUT_FACTOR = EVAL / "backtest_factor_ml2_desp_poscosecha.parquet"
OUT_FINAL = GOLD / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"


NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]
```

```
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _as_of_date() -> pd.Timestamp:
    return (pd.Timestamp.now().normalize() - pd.Timedelta(days=1))


def _latest_model(prefix: str) -> Path:
    files = sorted(MODELS.glob(f"{prefix}_*.pkl"))
    if not files:
        raise FileNotFoundError(f"No encuentro modelos en {MODELS} con prefijo {prefix}_*.pkl")
    return files[-1]


def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:
        if c in df.columns:
            return c
    raise KeyError("No encuentro fecha_post_pred_* en universe.")


def _resolve_factor_desp_ml1(df: pd.DataFrame) -> str:
    for c in ["factor_desp_ml1", "factor_desp_pred_ml1", "factor_desp"]:
        if c in df.columns:
            return c
    raise KeyError("No encuentro factor_desp ML1 en universe.")


def main(mode: str = "backtest", model_file: str | None = None) -> None:
    if mode not in {"backtest", "prod"}:
        raise ValueError("--mode debe ser backtest o prod")

    as_of = _as_of_date()

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Universe sin columnas: {sorted(miss)}")

    df["fecha"] = _to_date(df["fecha"])
    df = df.loc[df["fecha"].notna() & (df["fecha"] <= as_of)].copy()

    df["destino"] = _canon_str(df["destino"])
    if pd.api.types.is_numeric_dtype(df["grado"]):
        df["grado"] = _canon_int(df["grado"])
    else:
        df["grado"] = _canon_str(df["grado"])

    fpp = _resolve_fecha_post_pred(df)
    fd_ml1 = _resolve_factor_desp_ml1(df)

    df[fpp] = _to_date(df[fpp])
    df["fecha_post_pred_used"] = df[fpp]

    # Features calendario
    df["dow"] = df["fecha_post_pred_used"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha_post_pred_used"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha_post_pred_used"].dt.isocalendar().week.astype("Int64")
    # ensure cols exist
    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = pd.NA
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    X = df[NUM_COLS + CAT_COLS].copy()

    model_path = (MODELS / model_file) if model_file else _latest_model("desp_poscosecha_ml2")
    pipe = load(model_path)

    pred = pd.to_numeric(pd.Series(pipe.predict(X)), errors="coerce")
    # target era log_ratio_desp_clipped => pred es log_ratio_desp_pred
    df["log_ratio_desp_pred"] = pred

    # factor_desp_final = factor_desp_ml1 * exp(pred)
    df["factor_desp_ml1"] = pd.to_numeric(df[fd_ml1], errors="coerce")
    df["factor_desp_final_raw"] = df["factor_desp_ml1"] * np.exp(df["log_ratio_desp_pred"].astype(float))

    # clip factor final
    df["factor_desp_final"] = pd.to_numeric(df["factor_desp_final_raw"], errors="coerce").clip(lower=0.05, upp
er=1.00)

    # outputs
    EVAL.mkdir(parents=True, exist_ok=True)
    GOLD.mkdir(parents=True, exist_ok=True)

    fac = df[
        [
            "fecha",
            "fecha_post_pred_used",
            "bloque_base",
            "variedad_canon",
            "grado",
            "destino",
            "factor_desp_ml1",
            "log_ratio_desp_pred",
            "factor_desp_final",
        ]
    ].copy()
    fac["model_file"] = model_path.name
    fac["as_of_date"] = as_of
    fac["created_at"] = pd.Timestamp(datetime.now()).normalize()

    write_parquet(fac, OUT_FACTOR)
    write_parquet(df, OUT_FINAL)

    print(f"[OK] BACKTEST factor: {OUT_FACTOR} rows={len(fac):,}")
    print(f"[OK] BACKTEST final : {OUT_FINAL} rows={len(df):,}")
    print(f"     model={model_path.name} as_of_date={as_of.date()}")


if __name__ == "__main__":
    # argparse minimalista (sin depender de fire/typer)
    import argparse

    ap = argparse.ArgumentParser()
    ap.add_argument("--mode", default="backtest", choices=["backtest", "prod"])
    ap.add_argument("--model-file", default=None)
    args = ap.parse_args()

    main(mode=args.mode, model_file=args.model_file)

------------------------------------------------------------
[54/65] FILE: \src\models\ml2\train_ajuste_poscosecha_ml2.py
------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import hashlib

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import HistGradientBoostingRegressor

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
MODELS = DATA / "models" / "ml2"
EVAL = DATA / "eval" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_ajuste_poscosecha_ml2_v1.parquet"


def _hash8(s: str) -> str:
    return hashlib.sha1(s.encode("utf-8")).hexdigest()[:8]


def _safe_ohe() -> OneHotEncoder:
    # sklearn compatibility: sparse_output vs sparse
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)


def _wmae(y_true: np.ndarray, y_pred: np.ndarray, w: np.ndarray) -> float:
    y_true = np.asarray(y_true, dtype="float64")
    y_pred = np.asarray(y_pred, dtype="float64")
    w = np.asarray(w, dtype="float64")
    m = np.isfinite(y_true) & np.isfinite(y_pred) & np.isfinite(w) & (w >= 0)
    if not np.any(m):
        return float("nan")
    denom = float(np.sum(w[m]))
    if denom <= 0:
        return float("nan")
    return float(np.sum(np.abs(y_true[m] - y_pred[m]) * w[m]) / denom)


def main() -> None:
    df = read_parquet(IN_DS).copy()
    df.columns = [str(c).strip() for c in df.columns]

    # solo filas con real para entrenamiento
    m = df["factor_ajuste_real"].notna() & df["log_ratio_ajuste"].notna()
    d = df.loc[m].copy()

    if d.empty:
        raise ValueError("Dataset no tiene filas con real (factor_ajuste_real). No se puede entrenar.")

    # features
    num_cols = ["dow", "month", "weekofyear"]
    cat_cols = ["destino", "grado"]

    for c in num_cols:
        if c not in d.columns:
            d[c] = pd.NA
```

```
for c in cat_cols:
    if c not in d.columns:
        d[c] = "UNKNOWN"

X = d[num_cols + cat_cols].copy()
y = pd.to_numeric(d["log_ratio_ajuste"], errors="coerce").astype("float64").values
w = pd.to_numeric(d.get("w", 1.0), errors="coerce").fillna(0.0).astype("float64").values

# split temporal simple: 80/20 por fecha_post_pred_used
d["fecha_post_pred_used"] = pd.to_datetime(d["fecha_post_pred_used"], errors="coerce").dt.normalize()
order = d["fecha_post_pred_used"].fillna(pd.Timestamp("1970-01-01")).values
idx = np.argsort(order)

cut = int(0.80 * len(d))
tr_idx = idx[:cut]
va_idx = idx[cut:]

X_tr, X_va = X.iloc[tr_idx], X.iloc[va_idx]
y_tr, y_va = y[tr_idx], y[va_idx]
w_tr, w_va = w[tr_idx], w[va_idx]

pre = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", _safe_ohe(), cat_cols),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="squared_error",
    max_depth=6,
    learning_rate=0.08,
    max_iter=250,
    random_state=42,
)

pipe = Pipeline([("pre", pre), ("model", model)])
pipe.fit(X_tr, y_tr, model__sample_weight=w_tr)

pred_tr = pipe.predict(X_tr)
pred_va = pipe.predict(X_va)

mae_tr = _wmae(y_tr, pred_tr, w_tr)
mae_va = _wmae(y_va, pred_va, w_va)

# save
MODELS.mkdir(parents=True, exist_ok=True)
EVAL.mkdir(parents=True, exist_ok=True)

ts = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
tag = _hash8(f"{ts}|ajuste_poscosecha_ml2|{mae_va:.6f}")
model_name = f"ajuste_poscosecha_ml2_{ts}_{tag}.pkl"
meta_name = f"ajuste_poscosecha_ml2_{ts}_{tag}_meta.json"

model_path = MODELS / model_name
meta_path = MODELS / meta_name

dump(pipe, model_path)

meta = {
    "model_file": model_name,
    "created_at_utc": datetime.utcnow().isoformat(),
    "target": "log_ratio_ajuste = log(real/ml1) clipped",
    "clip_range_target": [-1.2, 1.2],
    "clip_range_apply": [0.50, 2.00],
    "features_num": num_cols,
    "features_cat": cat_cols,
    "n_train": int(len(tr_idx)),
    "n_val": int(len(va_idx)),
    "mae_log_train_w": float(mae_tr),
    "mae_log_val_w": float(mae_va),
    }

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

    out_eval = pd.DataFrame([{
        "model_file": model_name,
        "mae_log_train_w": mae_tr,
        "mae_log_val_w": mae_va,
        "n_train": int(len(tr_idx)),
        "n_val": int(len(va_idx)),
        "created_at": pd.Timestamp.utcnow(),
    }])
    eval_path = EVAL / "ml2_ajuste_poscosecha_train_val.parquet"
    write_parquet(out_eval, eval_path)

    print(f"[OK] Model saved: {model_path}")
    print(f"[OK] Meta  saved: {meta_path}")
    print(f"[OK] Eval  saved: {eval_path}")
    print(f"     MAE log train={mae_tr:.4f}  val={mae_va:.4f}")


if __name__ == "__main__":
    main()

----------------------------------------------------------
[55/65] FILE: \src\models\ml2\train_desp_poscosecha_ml2.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import hashlib
import json

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import HistGradientBoostingRegressor

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_desp_poscosecha_ml2_v1.parquet"
OUT_EVAL = EVAL / "ml2_desp_poscosecha_train_val.parquet"


NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]
TARGET_COL = "log_ratio_desp_clipped"
WEIGHT_COL = "tallos_w"


def _mae(a: np.ndarray, b: np.ndarray, w: np.ndarray | None = None) -> float:
    a = np.asarray(a, dtype=float)
    b = np.asarray(b, dtype=float)
    m = np.isfinite(a) & np.isfinite(b)
    if not np.any(m):
        return float("nan")
    err = np.abs(a[m] - b[m])
    if w is None:
        return float(np.mean(err))
    ww = np.asarray(w, dtype=float)[m]
    denom = float(np.sum(ww))
    if denom <= 0:
        return float(np.mean(err))
    return float(np.sum(err * ww) / denom)


def _fingerprint(df: pd.DataFrame) -> str:
    # hash liviano para versionado
    h = hashlib.sha256()
    h.update(str(df.shape).encode("utf-8"))
    h.update(str(df[TARGET_COL].dropna().describe().to_dict()).encode("utf-8"))
    return h.hexdigest()[:8]


def main() -> None:
    df = read_parquet(IN_DS).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = set(NUM_COLS + CAT_COLS + [TARGET_COL, WEIGHT_COL, "has_real", "fecha_post_pred_used"])
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Dataset sin columnas: {sorted(miss)}")

    d = df.loc[df["has_real"].astype(bool)].copy()
    d["fecha_post_pred_used"] = pd.to_datetime(d["fecha_post_pred_used"], errors="coerce").dt.normalize()

    # Split temporal: últimos 30 días (por fecha_post) a validación
    max_date = d["fecha_post_pred_used"].max()
    if pd.isna(max_date):
        raise ValueError("No hay fecha_post_pred_used válida en dataset con real.")

    cut = max_date - pd.Timedelta(days=30)

    tr = d.loc[d["fecha_post_pred_used"] < cut].copy()
    va = d.loc[d["fecha_post_pred_used"] >= cut].copy()

    X_tr = tr[NUM_COLS + CAT_COLS].copy()
    y_tr = pd.to_numeric(tr[TARGET_COL], errors="coerce").values
    w_tr = pd.to_numeric(tr[WEIGHT_COL], errors="coerce").fillna(0.0).values

    X_va = va[NUM_COLS + CAT_COLS].copy()
    y_va = pd.to_numeric(va[TARGET_COL], errors="coerce").values
    w_va = pd.to_numeric(va[WEIGHT_COL], errors="coerce").fillna(0.0).values

    # Preprocess: OHE denso (evita error sparse->dense)
    pre = ColumnTransformer(
        transformers=[
            ("cat", OneHotEncoder(handle_unknown="ignore", sparse_output=False), CAT_COLS),
            ("num", "passthrough", NUM_COLS),
        ],
        remainder="drop",
    )

    model = HistGradientBoostingRegressor(
        loss="squared_error",
        learning_rate=0.05,
        max_depth=6,
        max_iter=400,
        random_state=42,
    )

    pipe = Pipeline([("prep", pre), ("model", model)])

    pipe.fit(X_tr, y_tr, model__sample_weight=w_tr)

    pred_tr = pipe.predict(X_tr)
    pred_va = pipe.predict(X_va)
    mae_tr = _mae(y_tr, pred_tr, w_tr)
    mae_va = _mae(y_va, pred_va, w_va)

    # Persist
    MODELS.mkdir(parents=True, exist_ok=True)
    EVAL.mkdir(parents=True, exist_ok=True)

    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    fp = _fingerprint(d)
    model_name = f"desp_poscosecha_ml2_{ts}_{fp}.pkl"
    meta_name = f"desp_poscosecha_ml2_{ts}_{fp}_meta.json"

    dump(pipe, MODELS / model_name)

    meta = {
        "model_file": model_name,
        "dataset": str(IN_DS),
        "target": TARGET_COL,
        "clip_target": 1.2,
        "clip_factor_final": [0.05, 1.00],
        "split_cut_date": str(cut.date()),
        "mae_train": mae_tr,
        "mae_val": mae_va,
        "n_train": int(len(tr)),
        "n_val": int(len(va)),
        "created_at": ts,
    }

    with open(MODELS / meta_name, "w", encoding="utf-8") as f:
        json.dump(meta, f, ensure_ascii=False, indent=2)

    out_eval = pd.DataFrame(
        [
            {
                "split": "train",
                "n": int(len(tr)),
                "mae": mae_tr,
                "cut_date": str(cut.date()),
                "created_at": pd.Timestamp.now().normalize(),
            },
            {
                "split": "val",
                "n": int(len(va)),
                "mae": mae_va,
                "cut_date": str(cut.date()),
                "created_at": pd.Timestamp.now().normalize(),
            },
        ]
    )
    write_parquet(out_eval, OUT_EVAL)

    print(f"[OK] Model saved: {MODELS / model_name}")
    print(f"[OK] Meta  saved: {MODELS / meta_name}")
    print(f"[OK] Eval  saved: {OUT_EVAL}")
    print(f"     MAE log train={mae_tr:.4f}  val={mae_va:.4f}")


if __name__ == "__main__":
    main()

--------------------------------------------------------
[56/65] FILE: \src\models\ml2\train_dh_poscosecha_ml2.py
--------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import hashlib

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_dh_poscosecha_ml2_v1.parquet"

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]

TARGET = "err_dh_days_clipped"
WEIGHT_COL = "tallos_real"


def _run_id(prefix: str = "dh_poscosecha_ml2") -> str:
    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    h = hashlib.md5(ts.encode("utf-8")).hexdigest()[:8]
    return f"{prefix}_{ts}_{h}"


def _make_ohe_dense() -> OneHotEncoder:
    """
    sklearn >=1.2 usa sparse_output; sklearn <1.2 usa sparse.
    Forzamos salida densa para que HistGradientBoostingRegressor no falle.
    """
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)


def main() -> None:
    df = read_parquet(IN_DS).copy()
    df.columns = [str(c).strip() for c in df.columns]

    # Train solo donde hay real
    m = df["dh_real"].notna() & df["dh_ml1"].notna() & df[TARGET].notna()
    tr = df.loc[m].copy()
    if tr.empty:
        raise ValueError("No hay filas con real para entrenar DH ML2.")

    tr["fecha"] = pd.to_datetime(tr["fecha"], errors="coerce").dt.normalize()
    tr = tr.sort_values("fecha")

    # Split temporal: últimas 8 semanas val
    cut = tr["fecha"].max() - pd.Timedelta(days=56)
    is_val = tr["fecha"] >= cut

    train_df = tr.loc[~is_val].copy()
    val_df = tr.loc[is_val].copy()

    # fallback si val es muy pequeño
    if len(val_df) < 1000:
        n = len(tr)
        n_val = max(int(0.2 * n), 1000 if n >= 5000 else int(0.2 * n))
        val_df = tr.tail(n_val).copy()
        train_df = tr.iloc[: max(n - n_val, 1)].copy()

    # asegurar features
    for c in NUM_COLS:
        if c not in tr.columns:
            train_df[c] = pd.NA
            val_df[c] = pd.NA
    for c in CAT_COLS:
        if c not in tr.columns:
            train_df[c] = "UNKNOWN"
            val_df[c] = "UNKNOWN"

    X_train = train_df[NUM_COLS + CAT_COLS]
    y_train = pd.to_numeric(train_df[TARGET], errors="coerce")

    w_train = pd.to_numeric(train_df.get(WEIGHT_COL), errors="coerce").fillna(0.0).values.astype(float)
    w_train = np.where(w_train <= 0, 1.0, w_train)

    X_val = val_df[NUM_COLS + CAT_COLS]
    y_val = pd.to_numeric(val_df[TARGET], errors="coerce")

    w_val = pd.to_numeric(val_df.get(WEIGHT_COL), errors="coerce").fillna(0.0).values.astype(float)
    w_val = np.where(w_val <= 0, 1.0, w_val)

    pre = ColumnTransformer(
        transformers=[
            ("num", Pipeline(steps=[
                ("imputer", SimpleImputer(strategy="median")),
            ]), NUM_COLS),
            ("cat", Pipeline(steps=[
                ("imputer", SimpleImputer(strategy="most_frequent")),
                ("ohe", _make_ohe_dense()),  # <-- FIX: denso
            ]), CAT_COLS),
        ],
        remainder="drop",
        verbose_feature_names_out=False,
    )

    model = HistGradientBoostingRegressor(
        loss="absolute_error",
        max_depth=6,
        learning_rate=0.08,
        max_iter=300,
        random_state=42,
    )

    pipe = Pipeline(steps=[("pre", pre), ("model", model)])

    pipe.fit(X_train, y_train, model__sample_weight=w_train)

    pred_tr = pipe.predict(X_train)
    pred_va = pipe.predict(X_val)

    mae_tr = mean_absolute_error(y_train, pred_tr, sample_weight=w_train)
    mae_va = mean_absolute_error(y_val, pred_va, sample_weight=w_val)

    run_id = _run_id()

    MODELS.mkdir(parents=True, exist_ok=True)
    EVAL.mkdir(parents=True, exist_ok=True)

    out_model = MODELS / f"{run_id}.pkl"
    out_meta = MODELS / f"{run_id}_meta.json"
    out_eval = EVAL / "ml2_dh_poscosecha_train_val.parquet"

    dump(pipe, out_model)

    clip_err = float(pd.to_numeric(df.get("clip_err_days", pd.Series([5.0])).iloc[0], errors="coerce") or 5.0)

    meta = {
        "run_id": run_id,
        "created_at_utc": str(pd.Timestamp.utcnow()),
        "dataset": str(IN_DS),
        "target": TARGET,
        "clip_err_days": clip_err,
        "features_num": NUM_COLS,
        "features_cat": CAT_COLS,
        "split": {"type": "time", "val_window_days": 56},
        "metrics": {"mae_train_days": float(mae_tr), "mae_val_days": float(mae_va)},
        "note": "OHE forced dense for HistGradientBoostingRegressor compatibility",
    }

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

    ev = pd.DataFrame([{
        "run_id": run_id,
        "n_train": int(len(train_df)),
        "n_val": int(len(val_df)),
        "mae_train_days": float(mae_tr),
        "mae_val_days": float(mae_va),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])
    write_parquet(ev, out_eval)

    print(f"[OK] Model saved: {out_model}")
    print(f"[OK] Meta  saved: {out_meta}")
    print(f"[OK] Eval  saved: {out_eval}")
    print(f"     MAE days train={mae_tr:.4f}  val={mae_va:.4f}")


if __name__ == "__main__":
    main()

----------------------------------------------------------
[57/65] FILE: \src\models\ml2\train_harvest_horizon_ml2.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid

import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_DS = GOLD_DIR / "ml2_datasets" / "ds_harvest_horizon_ml2_v2.parquet"

# Guardrails
CLIP_ERROR_LO = -14
CLIP_ERROR_HI = 21
```

```
# Split temporal
VAL_QUANTILE = 0.80  # últimos 20% as_of_date como validación


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def main() -> None:
    df = read_parquet(IN_DS).copy()

    # Canon
    if "variedad_canon" in df.columns:
        df["variedad_canon"] = _canon_str(df["variedad_canon"])
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
    if "tipo_sp" in df.columns:
        df["tipo_sp"] = _canon_str(df["tipo_sp"])
    if "estado" in df.columns:
        df["estado"] = _canon_str(df["estado"])

    df["as_of_date"] = pd.to_datetime(df["as_of_date"], errors="coerce").dt.normalize()

    # Target
    y = pd.to_numeric(df["error_horizon_days"], errors="coerce")
    m = y.notna() & df["as_of_date"].notna()
    df = df.loc[m, :].copy()
    y = y.loc[m].astype(float)

    # Features: num + cat
    num_cols = [
        "tallos_proy",
        "days_sp_to_start_pred",
        "n_harvest_days_pred",
        "days_from_sp",
        "pred_error_start_days",
        "dow",
        "month",
        "weekofyear",
        "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
        "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
        "solar_cum_sp", "solar_7d",
        "temp_avg_7d",
    ]
    cat_cols = [
        "bloque_base",
        "variedad_canon",
        "area",
        "tipo_sp",
        "estado",
    ]

    # asegurar columnas existentes
    num_cols = [c for c in num_cols if c in df.columns]
    cat_cols = [c for c in cat_cols if c in df.columns]

    for c in num_cols:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    for c in cat_cols:
        df[c] = df[c].astype(str).fillna("")

    X = df[num_cols + cat_cols].copy()

    # Split temporal por as_of_date
    cutoff = df["as_of_date"].quantile(VAL_QUANTILE)
    is_val = df["as_of_date"] >= cutoff

    Xtr, ytr = X.loc[~is_val], y.loc[~is_val]
    Xva, yva = X.loc[is_val], y.loc[is_val]

    # OHE denso (compatible sklearn vieja)
    try:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

    pre = ColumnTransformer(
        transformers=[
            ("num", "passthrough", num_cols),
            ("cat", ohe, cat_cols),
        ],
        remainder="drop",
    )

    model = HistGradientBoostingRegressor(
        loss="absolute_error",
        max_depth=6,
        learning_rate=0.05,
        max_iter=300,
        random_state=42,
    )

    pipe = Pipeline([("pre", pre), ("model", model)])

    pipe.fit(Xtr, ytr)

    # Metrics
    pred_tr = pipe.predict(Xtr)
    pred_va = pipe.predict(Xva)

    mae_tr = float(mean_absolute_error(ytr, pred_tr))
    mae_va = float(mean_absolute_error(yva, pred_va))

    # Guardrail metrics (solo diagnóstico; el clip se aplica en APPLY)
    pred_va_clip = np.clip(pred_va, CLIP_ERROR_LO, CLIP_ERROR_HI)
    pct_clip_va = float(np.mean((pred_va <= CLIP_ERROR_LO) | (pred_va >= CLIP_ERROR_HI))) if len(pred_va) else
 np.nan
    mae_va_clip = float(mean_absolute_error(yva, pred_va_clip)) if len(pred_va) else np.nan

    run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]

    MODELS_DIR.mkdir(parents=True, exist_ok=True)
    EVAL_DIR.mkdir(parents=True, exist_ok=True)

    # Save model
    import joblib
    model_path = MODELS_DIR / f"harvest_horizon_ml2_{run_id}.pkl"
    joblib.dump(pipe, model_path)

    # Save meta
    meta = {
        "run_id": run_id,
        "created_at": datetime.now().isoformat(timespec="seconds"),
        "dataset": str(IN_DS),
        "target": "error_horizon_days = n_harvest_days_real - n_harvest_days_pred_ml1",
        "guardrails": {"clip_error_days": [CLIP_ERROR_LO, CLIP_ERROR_HI], "min_horizon_days": 7},
        "split": {"type": "temporal_quantile", "val_quantile": VAL_QUANTILE, "cutoff_as_of_date": str(cutoff.d
ate())},
        "num_cols": num_cols,
        "cat_cols": cat_cols,
        "metrics": {
            "mae_train": mae_tr,
            "mae_val": mae_va,
            "mae_val_clipped": mae_va_clip,
            "pct_clip_val": pct_clip_va,
            "n_train": int(len(ytr)),
            "n_val": int(len(yva)),
        },
    }
    meta_path = MODELS_DIR / f"harvest_horizon_ml2_{run_id}_meta.json"
    meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

    # Save eval row
    eval_row = pd.DataFrame([{
        "run_id": run_id,
        "cutoff_as_of_date": pd.to_datetime(cutoff).normalize(),
        "n_train": int(len(ytr)),
        "n_val": int(len(yva)),
        "mae_train": mae_tr,
        "mae_val": mae_va,
        "mae_val_clipped": mae_va_clip,
        "pct_clip_val": pct_clip_va,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])
    eval_path = EVAL_DIR / "ml2_harvest_horizon_train_cv.parquet"
    # append-safe: si existe, concatenar
    if eval_path.exists():
        old = read_parquet(eval_path)
        eval_row = pd.concat([old, eval_row], ignore_index=True)
    write_parquet(eval_row, eval_path)

    print(f"[OK] Model saved: {model_path}")
    print(f"[OK] Meta  saved: {meta_path}")
    print(f"[OK] Eval  saved: {eval_path}")
    print(f"     MAE train={mae_tr:.4f}  val={mae_va:.4f}  val_clipped={mae_va_clip:.4f}  pct_clip_val={pct_cl
ip_va:.3%}")


if __name__ == "__main__":
    main()

--------------------------------------------------------
[58/65] FILE: \src\models\ml2\train_harvest_start_ml2.py
--------------------------------------------------------
# src/models/ml2/train_harvest_start_ml2.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid
import numpy as np
import pandas as pd

from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_absolute_error
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

DS_PATH = GOLD_DIR / "ml2_datasets" / "ds_harvest_start_ml2_v2.parquet"
OUT_KPIS = EVAL_DIR / "ml2_harvest_start_train_cv.parquet"

TARGET = "error_start_days"

NUM_COLS = [
    "days_sp_to_start_pred",
    "n_harvest_days_pred",
    "tallos_proy",
    "dow", "month", "weekofyear",
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
    "temp_avg_7d",
]
CAT_COLS = [
    "bloque_base",
    "variedad_canon",
    "tipo_sp",
    "area",
    "estado",
]
CLIP_LO, CLIP_HI = -21, 21


def main() -> None:
    df = read_parquet(DS_PATH).copy()

    num_cols = [c for c in NUM_COLS if c in df.columns]
    cat_cols = [c for c in CAT_COLS if c in df.columns]

    df = df.sort_values("as_of_date").reset_index(drop=True)

    X = df[num_cols + cat_cols]
    y = df[TARGET].astype(float)

    # Preprocess
    try:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        # sklearn older
        ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

    pre = ColumnTransformer(
        transformers=[
            ("num", "passthrough", num_cols),
            ("cat", ohe, cat_cols),
        ],
        remainder="drop",
    )


    model = HistGradientBoostingRegressor(
        loss="absolute_error",
        max_depth=6,
        learning_rate=0.05,
        max_iter=400,
        random_state=42,
    )

    pipe = Pipeline(steps=[("pre", pre), ("model", model)])

    tscv = TimeSeriesSplit(n_splits=5)
    maes = []
    for fold, (tr, te) in enumerate(tscv.split(X), start=1):
        Xtr, Xte = X.iloc[tr], X.iloc[te]
        ytr, yte = y.iloc[tr], y.iloc[te]
        pipe.fit(Xtr, ytr)
        pred = pipe.predict(Xte)
        mae = float(mean_absolute_error(yte, pred))
        maes.append({"fold": fold, "mae_days": mae, "n_test": int(len(te))})

    pipe.fit(X, y)

    run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]
    MODELS_DIR.mkdir(parents=True, exist_ok=True)

    model_path = MODELS_DIR / f"harvest_start_ml2_{run_id}.pkl"
    meta_path = MODELS_DIR / f"harvest_start_ml2_{run_id}_meta.json"

    import joblib
    joblib.dump(pipe, model_path)

    meta = {
        "run_id": run_id,
        "trained_at": datetime.now().isoformat(),
        "dataset": str(DS_PATH),
        "target": TARGET,
        "num_cols": num_cols,
        "cat_cols": cat_cols,
        "cv": maes,
        "cv_mae_mean": float(np.mean([m["mae_days"] for m in maes])) if maes else None,
        "guardrails": {"clip_error_days": [CLIP_LO, CLIP_HI]},
    }
    meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

    # Guardar CV
    EVAL_DIR.mkdir(parents=True, exist_ok=True)
    kpi_df = pd.DataFrame(maes)
    kpi_df["run_id"] = run_id
    kpi_df["created_at"] = pd.Timestamp(datetime.now()).normalize()
    write_parquet(kpi_df, OUT_KPIS)

    print(f"[OK] Model saved: {model_path}")
    print(f"[OK] Meta saved : {meta_path}")
    print(f"[OK] CV KPIs saved: {OUT_KPIS}")


if __name__ == "__main__":
    main()

----------------------------------------------------------
[59/65] FILE: \src\models\ml2\train_hidr_poscosecha_ml2.py
----------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_hidr_poscosecha_ml2_v1.parquet"


def _ohe_dense() -> OneHotEncoder:
    # compat sklearn
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)


def main() -> None:
    df = read_parquet(IN_DS).copy()
    df.columns = [str(c).strip() for c in df.columns]
```

```
# solo filas con real
m = df.get("is_in_real", pd.Series(False, index=df.index)).astype(bool)
d = df.loc[m].copy()
if d.empty:
    raise ValueError("DS no tiene filas con real (is_in_real=1). No se puede entrenar.")

# features
num_cols = ["dow", "month", "weekofyear", "dh_dias_final"]
# dh_dias_final podría no existir en dataset; intenta alias
if "dh_dias_final" not in d.columns:
    for c in ["dh_dias_ml2", "dh_dias_ml1", "dh_dias"]:
        if c in d.columns:
            d = d.rename(columns={c: "dh_dias_final"})
            break
    if "dh_dias_final" not in d.columns:
        d["dh_dias_final"] = np.nan

cat_cols = ["destino", "grado", "variedad_canon"]

for c in num_cols:
    if c not in d.columns:
        d[c] = np.nan
for c in cat_cols:
    if c not in d.columns:
        d[c] = "UNKNOWN"

X = d[num_cols + cat_cols].copy()
y = pd.to_numeric(d["log_error_hidr_clipped"], errors="coerce")

# pesos por tallos (si no hay, 1)
w = pd.to_numeric(d.get("tallos_w"), errors="coerce").fillna(1.0).clip(lower=0.0)

# split simple por tiempo (más estable que random)
# tomamos las últimas ~20% fechas_post_pred como "val" si existe, si no random
split_col = None
for c in ["fecha_post_pred_final", "fecha_post_pred_ml2", "fecha_post_pred_ml1", "fecha_post_pred"]:
    if c in d.columns:
        split_col = c
        break

if split_col is not None:
    t = pd.to_datetime(d[split_col], errors="coerce").dt.normalize()
    q = t.quantile(0.80)
    train_idx = t <= q
else:
    rng = np.random.default_rng(42)
    train_idx = rng.random(len(d)) <= 0.80

X_train, y_train, w_train = X.loc[train_idx], y.loc[train_idx], w.loc[train_idx]
X_val, y_val, w_val = X.loc[~train_idx], y.loc[~train_idx], w.loc[~train_idx]

pre = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", _ohe_dense(), cat_cols),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="squared_error",
    max_depth=6,
    learning_rate=0.06,
    max_iter=350,
    min_samples_leaf=40,
    l2_regularization=0.0,
    random_state=42,
)

pipe = Pipeline([("pre", pre), ("model", model)])

pipe.fit(X_train, y_train, model__sample_weight=w_train)
```

```
# eval en log space
pred_tr = pipe.predict(X_train)
pred_va = pipe.predict(X_val)

def _mae(y_true, y_pred, w=None) -> float:
    a = pd.to_numeric(pd.Series(y_true), errors="coerce")
    b = pd.to_numeric(pd.Series(y_pred), errors="coerce")
    m = a.notna() & b.notna()
    if not m.any():
        return float("nan")

    err = (a[m] - b[m]).abs()

    if w is None:
        return float(err.mean())

    ww = pd.to_numeric(pd.Series(w), errors="coerce").fillna(0.0)
    ww = ww[m].clip(lower=0.0)
    denom = float(ww.sum())
    if denom <= 0:
        return float(err.mean())
    return float((err * ww).sum() / denom)

mae_tr = _mae(y_train, pred_tr, w_train)
mae_va = _mae(y_val, pred_va, w_val)

# artefactos
ts = datetime.now().strftime("%Y%m%d_%H%M%S")
uid = f"{np.random.default_rng().integers(0, 16**8):08x}"
model_name = f"hidr_poscosecha_ml2_{ts}_{uid}.pkl"
meta_name = f"hidr_poscosecha_ml2_{ts}_{uid}_meta.json"

MODELS.mkdir(parents=True, exist_ok=True)
EVAL.mkdir(parents=True, exist_ok=True)

dump(pipe, MODELS / model_name)

meta = {
    "model_file": model_name,
    "created_at_utc": pd.Timestamp.utcnow().isoformat(),
    "target": "log_error_hidr_clipped",
    "apply_clip_log": [-1.2, 1.2],
    "final_factor_clip": [0.60, 3.00],
    "features_num": num_cols,
    "features_cat": cat_cols,
    "mae_log_train_w": mae_tr,
    "mae_log_val_w": mae_va,
    "n_train": int(len(X_train)),
    "n_val": int(len(X_val)),
}
with open(MODELS / meta_name, "w", encoding="utf-8") as f:
    json.dump(meta, f, ensure_ascii=False, indent=2)

# guardar tabla simple train/val
out = pd.DataFrame([{
    "mae_log_train_w": mae_tr,
    "mae_log_val_w": mae_va,
    "n_train": int(len(X_train)),
    "n_val": int(len(X_val)),
    "model_file": model_name,
    "meta_file": meta_name,
    "created_at": pd.Timestamp.now().normalize(),
}])
write_parquet(out, EVAL / "ml2_hidr_poscosecha_train_val.parquet")

print(f"[OK] Model saved: {MODELS / model_name}")
print(f"[OK] Meta  saved: {MODELS / meta_name}")
print(f"[OK] Eval  saved: {EVAL / 'ml2_hidr_poscosecha_train_val.parquet'}")
print(f"     MAE log train(w)={mae_tr:.4f}  val(w)={mae_va:.4f}")
```

```
if __name__ == "__main__":
    main()

-----------------------------------------------------
[60/65] FILE: \src\models\ml2\train_peso_tallo_ml2.py
-----------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid

import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
MODELS = DATA / "models" / "ml2"
EVAL = DATA / "eval" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_peso_tallo_ml2_v1.parquet"

CLIP_LO, CLIP_HI = -0.8, 0.8
VAL_QUANTILE = 0.80


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def main() -> None:
    df = read_parquet(IN_DS).copy()

    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    for c in ["bloque_base", "variedad_canon", "grado", "tipo_sp", "estado", "area"]:
        if c in df.columns:
            df[c] = _canon_str(df[c])

    y = pd.to_numeric(df["log_error_peso"], errors="coerce")
    m = y.notna() & df["fecha"].notna()
    df = df.loc[m].copy()
    y = y.loc[m].astype(float).clip(CLIP_LO, CLIP_HI)

    # Features (num + cat)
    num_cols = [
        "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final",
        "peso_tallo_ml1_g",  # escala base
        # clima diario
        "gdc_dia", "rainfall_mm_dia", "en_lluvia_dia",
        "temp_avg_dia", "solar_energy_j_m2_dia",
        "wind_speed_avg_dia", "wind_run_dia",
        # calendario
        "dow", "month", "weekofyear",
    ]
    cat_cols = [
        "bloque_base", "variedad_canon", "grado",
        "area", "tipo_sp", "estado",
    ]

    num_cols = [c for c in num_cols if c in df.columns]
    cat_cols = [c for c in cat_cols if c in df.columns]

    for c in num_cols:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    for c in cat_cols:
        df[c] = df[c].astype(str).fillna("")

    X = df[num_cols + cat_cols].copy()

    cutoff = df["fecha"].quantile(VAL_QUANTILE)
    is_val = df["fecha"] >= cutoff

    Xtr, ytr = X.loc[~is_val], y.loc[~is_val]
    Xva, yva = X.loc[is_val], y.loc[is_val]

    try:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

    pre = ColumnTransformer(
        transformers=[
            ("num", "passthrough", num_cols),
            ("cat", ohe, cat_cols),
        ],
        remainder="drop",
    )

    model = HistGradientBoostingRegressor(
        loss="absolute_error",
        max_depth=6,
        learning_rate=0.05,
        max_iter=350,
        random_state=42,
    )

    pipe = Pipeline([("pre", pre), ("model", model)])
    pipe.fit(Xtr, ytr)

    pred_tr = pipe.predict(Xtr)
    pred_va = pipe.predict(Xva)

    mae_tr = float(mean_absolute_error(ytr, pred_tr))
    mae_va = float(mean_absolute_error(yva, pred_va))

    pred_va_clip = np.clip(pred_va, CLIP_LO, CLIP_HI)
    mae_va_clip = float(mean_absolute_error(yva, pred_va_clip))
    pct_clip_va = float(np.mean((pred_va <= CLIP_LO) | (pred_va >= CLIP_HI))) if len(pred_va) else np.nan

    run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]

    MODELS.mkdir(parents=True, exist_ok=True)
    EVAL.mkdir(parents=True, exist_ok=True)

    import joblib
    model_path = MODELS / f"peso_tallo_ml2_{run_id}.pkl"
    joblib.dump(pipe, model_path)

    meta = {
        "run_id": run_id,
        "created_at": datetime.now().isoformat(timespec="seconds"),
        "dataset": str(IN_DS),
        "target": "log_error_peso = log((peso_tallo_real_g+eps)/(peso_tallo_ml1_g+eps))",
        "guardrails": {"clip_log_error": [CLIP_LO, CLIP_HI]},
        "split": {"type": "temporal_quantile", "val_quantile": VAL_QUANTILE, "cutoff_fecha": str(pd.to_datetim
e(cutoff).date())},
        "num_cols": num_cols,
        "cat_cols": cat_cols,
        "metrics": {
            "mae_train_log": mae_tr,
            "mae_val_log": mae_va,
            "mae_val_log_clipped": mae_va_clip,
            "pct_clip_val": pct_clip_va,
            "n_train": int(len(ytr)),
            "n_val": int(len(yva)),
        },
    }
    meta_path = MODELS / f"peso_tallo_ml2_{run_id}_meta.json"
    meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

    eval_row = pd.DataFrame([{
        "run_id": run_id,
        "cutoff_fecha": pd.to_datetime(cutoff).normalize(),
        "n_train": int(len(ytr)),
        "n_val": int(len(yva)),
        "mae_train_log": mae_tr,
        "mae_val_log": mae_va,
        "mae_val_log_clipped": mae_va_clip,
        "pct_clip_val": pct_clip_va,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])
    eval_path = EVAL / "ml2_peso_tallo_train_cv.parquet"
    if eval_path.exists():
        old = read_parquet(eval_path)
        eval_row = pd.concat([old, eval_row], ignore_index=True)
    write_parquet(eval_row, eval_path)

    print(f"[OK] Model saved: {model_path}")
    print(f"[OK] Meta  saved: {meta_path}")
    print(f"[OK] Eval  saved: {eval_path}")
    print(f"     MAE log train={mae_tr:.4f}  val={mae_va:.4f}  val_clipped={mae_va_clip:.4f}  pct_clip_val={pc
t_clip_va:.3%}")


if __name__ == "__main__":
    main()

------------------------------------------------------
[61/65] FILE: \src\models\ml2\train_share_grado_ml2.py
------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid

import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
MODELS = DATA / "models" / "ml2"
EVAL = DATA / "eval" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_share_grado_ml2_v1.parquet"
```

```python
CLIP_LO, CLIP_HI = -1.2, 1.2
VAL_QUANTILE = 0.80


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def main() -> None:
    df = read_parquet(IN_DS).copy()

    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    for c in ["bloque_base", "variedad_canon", "grado", "tipo_sp", "estado", "area"]:
        if c in df.columns:
            df[c] = _canon_str(df[c])

    y = pd.to_numeric(df["log_error_share"], errors="coerce")
    m = y.notna() & df["fecha"].notna()
    df = df.loc[m].copy()
    y = y.loc[m].astype(float).clip(CLIP_LO, CLIP_HI)

    # Features
    num_cols = [
        "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final",
        # contexto de volumen (importante para shares)
        "tallos_pred_ml1_grado_dia",
        "share_grado_ml1",
        # clima
        "gdc_dia", "rainfall_mm_dia", "en_lluvia_dia",
        "temp_avg_dia", "solar_energy_j_m2_dia",
        "wind_speed_avg_dia", "wind_run_dia",
        # calendario
        "dow", "month", "weekofyear",
    ]
    cat_cols = [
        "grado",
        "variedad_canon",
        "area", "tipo_sp", "estado",
        # bloque_base puede overfit fuerte; lo dejamos pero si quieres lo quitamos después
        "bloque_base",
    ]

    num_cols = [c for c in num_cols if c in df.columns]
    cat_cols = [c for c in cat_cols if c in df.columns]

    for c in num_cols:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    for c in cat_cols:
        df[c] = df[c].astype(str).fillna("")

    X = df[num_cols + cat_cols].copy()

    cutoff = df["fecha"].quantile(VAL_QUANTILE)
    is_val = df["fecha"] >= cutoff

    Xtr, ytr = X.loc[~is_val], y.loc[~is_val]
    Xva, yva = X.loc[is_val], y.loc[is_val]

    try:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

    pre = ColumnTransformer(
        transformers=[
            ("num", "passthrough", num_cols),
            ("cat", ohe, cat_cols),
        ],
        remainder="drop",
    )

    model = HistGradientBoostingRegressor(
        loss="absolute_error",
        max_depth=6,
        learning_rate=0.05,
        max_iter=350,
        random_state=42,
    )

    pipe = Pipeline([("pre", pre), ("model", model)])
    pipe.fit(Xtr, ytr)

    pred_tr = pipe.predict(Xtr)
    pred_va = pipe.predict(Xva)

    mae_tr = float(mean_absolute_error(ytr, pred_tr))
    mae_va = float(mean_absolute_error(yva, pred_va))

    pred_va_clip = np.clip(pred_va, CLIP_LO, CLIP_HI)
    mae_va_clip = float(mean_absolute_error(yva, pred_va_clip))
    pct_clip_va = float(np.mean((pred_va <= CLIP_LO) | (pred_va >= CLIP_HI))) if len(pred_va) else np.nan

    run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]

    MODELS.mkdir(parents=True, exist_ok=True)
    EVAL.mkdir(parents=True, exist_ok=True)

    import joblib
    model_path = MODELS / f"share_grado_ml2_{run_id}.pkl"
    joblib.dump(pipe, model_path)

    meta = {
        "run_id": run_id,
        "created_at": datetime.now().isoformat(timespec="seconds"),
        "dataset": str(IN_DS),
        "target": "log_error_share = log((share_real+eps)/(share_ml1+eps))",
        "guardrails": {"clip_log_error": [CLIP_LO, CLIP_HI]},
        "split": {
            "type": "temporal_quantile",
            "val_quantile": VAL_QUANTILE,
            "cutoff_fecha": str(pd.to_datetime(cutoff).date()),
        },
        "num_cols": num_cols,
        "cat_cols": cat_cols,
        "metrics": {
            "mae_train_log": mae_tr,
            "mae_val_log": mae_va,
            "mae_val_log_clipped": mae_va_clip,
            "pct_clip_val": pct_clip_va,
            "n_train": int(len(ytr)),
            "n_val": int(len(yva)),
        },
    }
    meta_path = MODELS / f"share_grado_ml2_{run_id}_meta.json"
    meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

    eval_row = pd.DataFrame([{
        "run_id": run_id,
        "cutoff_fecha": pd.to_datetime(cutoff).normalize(),
        "n_train": int(len(ytr)),
        "n_val": int(len(yva)),
        "mae_train_log": mae_tr,
        "mae_val_log": mae_va,
        "mae_val_log_clipped": mae_va_clip,
        "pct_clip_val": pct_clip_va,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])
    eval_path = EVAL / "ml2_share_grado_train_cv.parquet"
    if eval_path.exists():
        old = read_parquet(eval_path)
        eval_row = pd.concat([old, eval_row], ignore_index=True)
    write_parquet(eval_row, eval_path)

    print(f"[OK] Model saved: {model_path}")
```

```
    print(f"[OK] Meta  saved: {meta_path}")
    print(f"[OK] Eval  saved: {eval_path}")
    print(f"     MAE log train={mae_tr:.4f}  val={mae_va:.4f}  val_clipped={mae_va_clip:.4f}  pct_clip_val={pc
t_clip_va:.3%}")


if __name__ == "__main__":
    main()

-------------------------------------------------------
[62/65] FILE: \src\models\ml2\train_tallos_curve_ml2.py
-------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid

import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet


def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]


ROOT = _project_root()
DATA_DIR = ROOT / "data"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_DS = GOLD_DIR / "ml2_datasets" / "ds_tallos_curve_ml2_v2.parquet"

# Guardrails (mismo rango del dataset builder)
CLIP_LOG_ERR_LO = -1.5
CLIP_LOG_ERR_HI = 1.5

VAL_QUANTILE = 0.80  # último 20% fechas para validación


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def main() -> None:
    df = read_parquet(IN_DS).copy()

    # Canon + fecha
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
    if "variedad_canon" in df.columns:
        df["variedad_canon"] = _canon_str(df["variedad_canon"])
    if "tipo_sp" in df.columns:
        df["tipo_sp"] = _canon_str(df["tipo_sp"])
    if "estado" in df.columns:
        df["estado"] = _canon_str(df["estado"])

    # Target
    y = pd.to_numeric(df["log_error"], errors="coerce")
    m = y.notna() & df["fecha"].notna()
    df = df.loc[m, :].copy()
```

```
y = y.loc[m].astype(float).clip(CLIP_LOG_ERR_LO, CLIP_LOG_ERR_HI)

# Features (num + cat)
num_cols = [
    "rel_pos_final",
    "day_in_harvest_final",
    "n_harvest_days_final",
    "tallos_pred_ml1_dia",
    "tallos_real_dia",
    # clima día
    "gdc_dia",
    "rainfall_mm_dia",
    "en_lluvia_dia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    # calendario
    "dow", "month", "weekofyear",
]
cat_cols = [
    "bloque_base",
    "variedad_canon",
    "area",
    "tipo_sp",
    "estado",
]

num_cols = [c for c in num_cols if c in df.columns]
cat_cols = [c for c in cat_cols if c in df.columns]

# Limpieza
for c in num_cols:
    df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
for c in cat_cols:
    df[c] = df[c].astype(str).fillna("")

X = df[num_cols + cat_cols].copy()

# Split temporal por fecha
cutoff = df["fecha"].quantile(VAL_QUANTILE)
is_val = df["fecha"] >= cutoff

Xtr, ytr = X.loc[~is_val], y.loc[~is_val]
Xva, yva = X.loc[is_val], y.loc[is_val]

# OHE denso
try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

pre = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", ohe, cat_cols),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="absolute_error",
    max_depth=6,
    learning_rate=0.05,
    max_iter=350,
    random_state=42,
)

pipe = Pipeline([("pre", pre), ("model", model)])
pipe.fit(Xtr, ytr)

# Métricas sobre log_error
```

```
    pred_tr = pipe.predict(Xtr)
    pred_va = pipe.predict(Xva)

    mae_tr = float(mean_absolute_error(ytr, pred_tr))
    mae_va = float(mean_absolute_error(yva, pred_va))

    # Diagnóstico: % clipping si se aplica (aún no lo aplicamos aquí)
    pred_va_clip = np.clip(pred_va, CLIP_LOG_ERR_LO, CLIP_LOG_ERR_HI)
    pct_clip_va = float(np.mean((pred_va <= CLIP_LOG_ERR_LO) | (pred_va >= CLIP_LOG_ERR_HI))) if len(pred_va)
else np.nan
    mae_va_clip = float(mean_absolute_error(yva, pred_va_clip)) if len(pred_va) else np.nan

    run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]

    MODELS_DIR.mkdir(parents=True, exist_ok=True)
    EVAL_DIR.mkdir(parents=True, exist_ok=True)

    # Save model
    import joblib
    model_path = MODELS_DIR / f"tallos_curve_ml2_{run_id}.pkl"
    joblib.dump(pipe, model_path)

    # Save meta
    meta = {
        "run_id": run_id,
        "created_at": datetime.now().isoformat(timespec="seconds"),
        "dataset": str(IN_DS),
        "target": "log_error = log((tallos_real_dia+eps)/(tallos_pred_ml1_dia+eps))",
        "guardrails": {"clip_log_error": [CLIP_LOG_ERR_LO, CLIP_LOG_ERR_HI]},
        "split": {"type": "temporal_quantile", "val_quantile": VAL_QUANTILE, "cutoff_fecha": str(pd.to_datetim
e(cutoff).date())},
        "num_cols": num_cols,
        "cat_cols": cat_cols,
        "metrics": {
            "mae_train_log": mae_tr,
            "mae_val_log": mae_va,
            "mae_val_log_clipped": mae_va_clip,
            "pct_clip_val": pct_clip_va,
            "n_train": int(len(ytr)),
            "n_val": int(len(yva)),
        },
    }
    meta_path = MODELS_DIR / f"tallos_curve_ml2_{run_id}_meta.json"
    meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

    # Save eval row
    eval_row = pd.DataFrame([{
        "run_id": run_id,
        "cutoff_fecha": pd.to_datetime(cutoff).normalize(),
        "n_train": int(len(ytr)),
        "n_val": int(len(yva)),
        "mae_train_log": mae_tr,
        "mae_val_log": mae_va,
        "mae_val_log_clipped": mae_va_clip,
        "pct_clip_val": pct_clip_va,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }])
    eval_path = EVAL_DIR / "ml2_tallos_curve_train_cv.parquet"
    if eval_path.exists():
        old = read_parquet(eval_path)
        eval_row = pd.concat([old, eval_row], ignore_index=True)
    write_parquet(eval_row, eval_path)

    print(f"[OK] Model saved: {model_path}")
    print(f"[OK] Meta  saved: {meta_path}")
    print(f"[OK] Eval  saved: {eval_path}")
    print(f"     MAE log train={mae_tr:.4f}  val={mae_va:.4f}  val_clipped={mae_va_clip:.4f}  pct_clip_val={pc
t_clip_va:.3%}")


if __name__ == "__main__":
    main()
```

```
--------------------------------------------------------------------
[63/65] FILE: \src\preds\build_pred_oferta_dia_from_universe_ml1.py
--------------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet


IN_GRID = Path("data/gold/universe_harvest_grid_ml1.parquet")
IN_MAESTRO = Path("data/silver/fact_ciclo_maestro.parquet")

OUT = Path("data/preds/pred_oferta_dia.parquet")


def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")


def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()


def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Cols={list(df.columns)}")


def main() -> None:
    created_at = datetime.now().isoformat(timespec="seconds")

    if not IN_GRID.exists():
        raise FileNotFoundError(f"No existe: {IN_GRID}")
    if not IN_MAESTRO.exists():
        raise FileNotFoundError(f"No existe: {IN_MAESTRO}")

    grid = read_parquet(IN_GRID).copy()
    maestro = read_parquet(IN_MAESTRO).copy()

    # -------------------------
    # Requisitos mínimos
    # -------------------------
    _require(grid, ["ciclo_id", "fecha"], "universe_harvest_grid_ml1")
    _require(maestro, ["ciclo_id", "tallos_proy"], "fact_ciclo_maestro")

    # -------------------------
    # Canon
    # -------------------------
    grid["ciclo_id"] = grid["ciclo_id"].astype(str)
    grid["fecha"] = _to_date(grid["fecha"])

    # bloque_base / variedad_canon desde grid (preferido)
    if "bloque_base" in grid.columns:
        grid["bloque_base"] = _canon_int(grid["bloque_base"])
    elif "bloque_padre" in grid.columns:
        grid["bloque_base"] = _canon_int(grid["bloque_padre"])

    if "variedad_canon" in grid.columns:
        grid["variedad_canon"] = _canon_str(grid["variedad_canon"])
    elif "variedad" in grid.columns:
        grid["variedad_canon"] = _canon_str(grid["variedad"])

    maestro["ciclo_id"] = maestro["ciclo_id"].astype(str)
    maestro["tallos_proy"] = pd.to_numeric(maestro["tallos_proy"], errors="coerce").astype(float)

    # opcionales de maestro
    for c in ["bloque_base", "bloque", "bloque_padre"]:
        if c in maestro.columns:
            maestro[c] = _canon_int(maestro[c])
    for c in ["variedad_canon", "variedad"]:
        if c in maestro.columns:
            maestro[c] = _canon_str(maestro[c])
    if "fecha_sp" in maestro.columns:
        maestro["fecha_sp"] = _to_date(maestro["fecha_sp"])
    for c in ["fecha_inicio_cosecha", "fecha_fin_cosecha"]:
        if c in maestro.columns:
            maestro[c] = _to_date(maestro[c])

    # -------------------------
    # Dedupe hard del grid
    # -------------------------
    key = ["ciclo_id", "fecha"]
    # Si tu grid trae bloque_base/variedad_canon, las metemos al grano final
    if "bloque_base" in grid.columns:
        key.append("bloque_base")
    if "variedad_canon" in grid.columns:
        key.append("variedad_canon")

    dup = int(grid.duplicated(subset=key).sum())
    if dup > 0:
        # en grid, duplicados deberían ser imposibles; los colapsamos por seguridad
        print(f"[WARN] universe_harvest_grid_ml1 duplicado por {key}; colapso. dup={dup:,}")
        agg = {}
        for c in ["harvest_start_pred", "harvest_end_pred", "n_harvest_days_pred"]:
            if c in grid.columns:
                agg[c] = "first"
        for c in ["area", "tipo_sp", "estado", "bloque", "bloque_padre"]:
            if c in grid.columns and c not in key:
                agg[c] = "first"
        grid = grid.groupby(key, as_index=False).agg(agg) if agg else grid.drop_duplicates(subset=key)

    # -------------------------
    # Join maestro (tallos_proy y metadatos)
    # -------------------------
    m_take = ["ciclo_id", "tallos_proy"]
    for c in ["bloque", "bloque_padre", "bloque_base", "variedad", "variedad_canon", "tipo_sp", "area", "estad
o"]:
        if c in maestro.columns:
            m_take.append(c)
    for c in ["fecha_sp", "fecha_inicio_cosecha", "fecha_fin_cosecha"]:
        if c in maestro.columns:
            m_take.append(c)

    m2 = maestro[m_take].drop_duplicates(subset=["ciclo_id"])
    out = grid.merge(m2, on="ciclo_id", how="left", suffixes=("", "_m"))

    # asegurar bloque_base/variedad_canon
    if "bloque_base" not in out.columns and "bloque_base_m" in out.columns:
        out["bloque_base"] = out["bloque_base_m"]
    if "variedad_canon" not in out.columns:
        if "variedad_canon_m" in out.columns:
            out["variedad_canon"] = out["variedad_canon_m"]
        elif "variedad_m" in out.columns:
            out["variedad_canon"] = out["variedad_m"]

    # Ventana: preferimos la predicha del grid
    if "harvest_start_pred" in out.columns:
        out["harvest_start"] = _to_date(out["harvest_start_pred"])
    elif "fecha_inicio_cosecha" in out.columns:
        out["harvest_start"] = _to_date(out["fecha_inicio_cosecha"])
    else:
        out["harvest_start"] = pd.NaT

    if "harvest_end_pred" in out.columns:
        out["harvest_end_eff"] = _to_date(out["harvest_end_pred"])
    elif "fecha_fin_cosecha" in out.columns:
        out["harvest_end_eff"] = _to_date(out["fecha_fin_cosecha"])
    else:
        out["harvest_end_eff"] = pd.NaT

    if "n_harvest_days_pred" in out.columns:
        out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days_pred"], errors="coerce").astype("Int64")
    else:
        # fallback por diferencia
        out["n_harvest_days"] = (out["harvest_end_eff"] - out["harvest_start"]).dt.days.add(1)
        out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days"], errors="coerce").astype("Int64")

    # -------------------------
    # Oferta baseline: uniform + ajuste de residuo último día
    # -------------------------
    out["tallos_proy"] = pd.to_numeric(out["tallos_proy"], errors="coerce")
    out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days"], errors="coerce").astype("Int64")

    bad = out["tallos_proy"].isna() | out["n_harvest_days"].isna() | (out["n_harvest_days"].astype(float) <= 0
)
    if bad.any():
        nbad = int(bad.sum())
        print(f"[WARN] filas sin tallos_proy o n_harvest_days inválido: {nbad:,}. tallos_pred=0 en esas filas.
")
        out.loc[bad, "tallos_pred"] = 0.0

    ok = ~bad
    out.loc[ok, "tallos_pred"] = out.loc[ok, "tallos_proy"] / out.loc[ok, "n_harvest_days"].astype(float)

    # ajuste de residuo para garantizar sum exacto por ciclo
    grp = ["ciclo_id"]
    out = out.sort_values(["ciclo_id", "fecha"]).reset_index(drop=True)

    # marca último día por ciclo dentro del grid
    out["_is_last"] = out["fecha"].eq(out.groupby("ciclo_id")["fecha"].transform("max"))

    sums = out.groupby("ciclo_id", dropna=False)["tallos_pred"].transform("sum")
    target = out.groupby("ciclo_id", dropna=False)["tallos_proy"].transform("max")

    resid = (target - sums)
    out.loc[out["_is_last"] & ok, "tallos_pred"] = out.loc[out["_is_last"] & ok, "tallos_pred"] + resid[out["_
is_last"] & ok]
    out = out.drop(columns=["_is_last"], errors="ignore")

    # stage fijo (este dataset ES harvest-grid)
    out["stage"] = "HARVEST"

    # -------------------------
    # Salida final
    # -------------------------
    out["created_at"] = created_at

    cols = [
        "ciclo_id", "fecha",
        "bloque" if "bloque" in out.columns else None,
        "bloque_padre" if "bloque_padre" in out.columns else None,
        "bloque_base",
        "variedad" if "variedad" in out.columns else None,
        "variedad_canon",
        "tipo_sp" if "tipo_sp" in out.columns else None,
        "area" if "area" in out.columns else None,
        "estado" if "estado" in out.columns else None,
        "stage",
        "harvest_start",
        "harvest_end_eff",
        "n_harvest_days",
        "tallos_proy",
        "tallos_pred",
        "created_at",
    ]
    cols = [c for c in cols if c is not None and c in out.columns]
    out = out[cols].sort_values(["ciclo_id", "fecha", "bloque_base", "variedad_canon"]).reset_index(drop=True)
    # Checks finales
    key2 = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    dup2 = int(out.duplicated(subset=key2).sum())
    if dup2:
        raise ValueError(f"[FATAL] Salida tiene duplicados por {key2}: {dup2}")

    cyc = out.groupby("ciclo_id", dropna=False).agg(
        proy=("tallos_proy", "max"),
        sum_pred=("tallos_pred", "sum"),
    ).reset_index()
    cyc["abs_diff"] = (cyc["proy"] - cyc["sum_pred"]).abs()
    print(f"[CHECK] ciclo mass-balance | max abs diff: {float(cyc['abs_diff'].max()):.12f}")

    OUT.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(out, OUT)
    print(f"OK -> {OUT} | rows={len(out):,} | fecha_min={out['fecha'].min().date()} fecha_max={out['fecha'].ma
x().date()}")


if __name__ == "__main__":
    main()

----------------------------------------------------------------
[64/65] FILE: \src\silver\build_ciclo_maestro_from_fenograma.py
----------------------------------------------------------------
# src/silver/build_ciclo_maestro_from_fenograma.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import re
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet
from common.ids import make_bloque_id, make_variedad_id, make_ciclo_id
from common.timegrid import build_grid_ciclo_fecha


# -------------------------
# Helpers
# -------------------------
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)


def normalizar_columnas(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c).strip() for c in df.columns]
    return df


def to_datetime_safe(x):
    return pd.to_datetime(x, errors="coerce")


def to_numeric_safe(x):
    return pd.to_numeric(x, errors="coerce")


def normalize_date_col(df: pd.DataFrame, col: str):
    df = df.copy()
    df[col] = to_datetime_safe(df[col]).dt.normalize()
    return df


def reemplazos_area(series: pd.Series) -> pd.Series:
    return series.astype(str).replace({"SSJ": "A-4", "MM1": "MH1", "MM2": "MH2"})
```

```
# -------------------------
# BRONZE RAW readers (indices: col_0..col_n)
# -------------------------
def _strip_cell(x) -> str:
    if x is None or (isinstance(x, float) and np.isnan(x)):
        return ""
    return str(x).strip()


def _raw_has_col0(df: pd.DataFrame) -> bool:
    return any(str(c).strip().lower() == "col_0" for c in df.columns)


def _ensure_cols_are_strings(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c) for c in df.columns]
    return df


def _rebuild_header_like_xl(raw: pd.DataFrame) -> pd.DataFrame:
    """
    Replica el patrón de tu fenograma_historia_xl() original:
      raw = raw[~raw[0].isna() & (raw[0].astype(str).str.strip() != "")].copy()
      raw = raw.iloc[1:].copy()
      raw.columns = raw.iloc[0]
      df = raw.iloc[1:].copy()

    En Bronze: raw viene como col_0..col_n (todo string).
    """
    raw = _ensure_cols_are_strings(raw)

    col0 = "col_0" if "col_0" in raw.columns else raw.columns[0]
    rr = raw.copy()

    mask = rr[col0].notna() & (rr[col0].astype(str).str.strip() != "")
    rr = rr[mask].copy()

    # drop first row
    if len(rr) < 2:
        return pd.DataFrame()

    rr = rr.iloc[1:].copy()

    # header row is first row now
    header = rr.iloc[0].tolist()
    header = [re.sub(r"\s+", " ", _strip_cell(h).replace("\n", " ").replace("\r", " ")).strip() for h in heade
r]
    rr = rr.iloc[1:].copy()
    rr.columns = header

    rr = normalizar_columnas(rr)
    return rr


def _rebuild_header_like_clo(raw: pd.DataFrame) -> pd.DataFrame:
    """
    Replica el patrón de tu fenograma_historia_clo() original:
      raw = raw[~raw[0].isna() & (raw[0].astype(str).str.strip() != "")].copy()
      raw.columns = raw.iloc[0]
      df = raw.iloc[1:].copy()

    En Bronze: raw viene como col_0..col_n (todo string).
    """
    raw = _ensure_cols_are_strings(raw)
    col0 = "col_0" if "col_0" in raw.columns else raw.columns[0]
    rr = raw.copy()

    mask = rr[col0].notna() & (rr[col0].astype(str).str.strip() != "")
    rr = rr[mask].copy()

    if len(rr) < 2:
        return pd.DataFrame()

    header = rr.iloc[0].tolist()
    header = [re.sub(r"\s+", " ", _strip_cell(h).replace("\n", " ").replace("\r", " ")).strip() for h in heade
r]
    rr = rr.iloc[1:].copy()
    rr.columns = header

    rr = normalizar_columnas(rr)
    return rr


# -------------------------
# Fenograma activo (tu lógica)
# -------------------------
def fenograma_activo(df_fenograma_xlsm: pd.DataFrame, bal: pd.DataFrame, hoy: pd.Timestamp) -> pd.DataFrame:
    df = normalizar_columnas(df_fenograma_xlsm)

    if "Area " in df.columns and "Area" not in df.columns:
        df = df.rename(columns={"Area ": "Area"})

    df["Bloques"] = df["Bloques"].astype(str)

    valores_excluir = {
        None, 0, 500, 625, 750, 875, 1000, 6025,
        "%1000 GR", "%350 GR", "%500 GR", "%750 GR",
        "00 a 20", "21 a 40", "41 a 60", "61 a 80", "81 a 100",
        "ANDES", "Cajas por cosechar entre:", "CAMPO", "CLOUD", "CV",
        "Distribucion de cosecha en el campo", "DISTRIBUCION POR GRADOS",
        "DOMINGO", "FLOR GRANDE = FG", "FLOR PEQUEÑA = FP", "GENERAL",
        "JUEVES", "LUNES", "MARTES", "MH1", "MH2", "MIERCOLES",
        "MILLION CLOUD", "MM1", "MM2", "OTRO",
        "PROYECCION CAJAS POR GRADOS",
        "SABADO", "SI", "SJP", "SSJ", "Tallos/caja",
        "TOTAL", "Total", "Total cajas", "TOTAL GYPSOS",
        "Total tallos", "VENTAS", "VIERNES", "x",
        "XLENCE", "XLENCE FIVE STARS"
    }

    if "Fiesta" in df.columns:
        df = df[~df["Fiesta"].isin(valores_excluir)].copy()

    df = df[df["Bloques"].notna()].copy()

    cols = ["Bloques", "Fecha S/P", "Area", "S/P", "Variedad", "Tallos / Bloque"]
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"fenograma_activo: faltan columnas esperadas en fenograma_xlsm_raw: {missing}")

    df = df[cols].copy()
    df["Fecha S/P"] = to_datetime_safe(df["Fecha S/P"])
    df["Tallos / Bloque"] = to_numeric_safe(df["Tallos / Bloque"]).fillna(0)

    agg = (
        df.groupby(["Bloques", "S/P", "Variedad", "Area"], dropna=False)
          .agg(**{
              "Fecha S/P": ("Fecha S/P", "max"),
              "Tallos_Proy": ("Tallos / Bloque", "sum")
          })
          .reset_index()
    )

    agg = agg[agg["Area"].notna()].copy()
    agg = agg[(agg["Area"] != 0) & (agg["Area"] != "0")].copy()
    agg["Area"] = reemplazos_area(agg["Area"])

    # Join con balanza para calcular días vegetativos
    tmp = agg.merge(bal, left_on="Bloques", right_on="Bloque", how="left").drop(columns=["Bloque"])
    tmp["Personalizado"] = (tmp["Fecha"] - tmp["Fecha S/P"]).dt.days

    tmp_f = tmp[(tmp["Personalizado"] > 30) | (tmp["Personalizado"].isna())].copy()
    g = (
        tmp_f.groupby(["Bloques", "S/P", "Fecha S/P"], dropna=False)
             .agg(
                 Dias_Vegetativo=("Personalizado", "min"),
                 Dias_Vegetativo_1=("Personalizado", "max")
             )
             .reset_index()
    )

    g["Fecha_Inicio_Cosecha (Primer Tallo)"] = g["Fecha S/P"] + pd.to_timedelta(g["Dias_Vegetativo"], unit="D"
)
    g.loc[g["Dias_Vegetativo"].isna(), "Fecha_Inicio_Cosecha (Primer Tallo)"] = pd.NaT

    fin_calc = g["Fecha S/P"] + pd.to_timedelta(g["Dias_Vegetativo_1"], unit="D")
    g["Fecha_Fin_Cosecha"] = fin_calc

    cond1 = g["Dias_Vegetativo"].isna()
    cond2 = g["Dias_Vegetativo_1"] <= g["Dias_Vegetativo"]
    cond3 = fin_calc >= (hoy - pd.Timedelta(days=4))
    g.loc[cond1 | cond2 | cond3, "Fecha_Fin_Cosecha"] = pd.NaT

    base = agg.merge(
        g[["Bloques", "Fecha S/P", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha"]],
        on=["Bloques", "Fecha S/P"],
        how="left"
    )

    base = base[base["Fecha S/P"] < hoy].copy()
    base["Estado"] = "ACTIVO"

    base = normalize_date_col(base, "Fecha S/P")
    base = normalize_date_col(base, "Fecha_Inicio_Cosecha (Primer Tallo)")
    base = normalize_date_col(base, "Fecha_Fin_Cosecha")

    return base[[
        "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
        "Tallos_Proy", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
    ]]


# -------------------------
# Históricos XL/CLO (desde BRONZE raw indices_*.parquet)
# -------------------------
def fenograma_historia_xl(raw_indices_xl: pd.DataFrame, fecha_min_hist: pd.Timestamp) -> pd.DataFrame:
    df = _rebuild_header_like_xl(raw_indices_xl)
    if df.empty:
        return pd.DataFrame(columns=[
            "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
            "Tallos_Proy", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
        ])

    # Filtros Pruebas (idéntico a tu lógica)
    if "Pruebas" in df.columns:
        df = df[df["Pruebas"].isna()].copy()
    elif "Pruebas " in df.columns:
        df = df[df["Pruebas "].isna()].copy()

    # Parse fechas
    if "fecha se s/p" not in df.columns:
        raise ValueError("indices_xl_raw: no encuentro columna 'fecha se s/p' tras reconstruir headers.")
    df["fecha se s/p"] = to_datetime_safe(df["fecha se s/p"])
    df["Fecha FIN cos"] = to_datetime_safe(df.get("Fecha FIN cos"))
    df["Fecha Inc cos"] = to_datetime_safe(df.get("Fecha Inc cos"))

    df = df[df["fecha se s/p"].notna()].copy()
    df = df[df["fecha se s/p"] >= fecha_min_hist].copy()

    needed = ["AREA_PRODUCT", "fecha se s/p", "Fecha Inc cos", "Fecha FIN cos", "Bloque", "p/s", "TALLOS TOTAL
ES EN VERDE"]
    missing = [c for c in needed if c not in df.columns]
    if missing:
        raise ValueError(f"indices_xl_raw: faltan columnas requeridas: {missing}")
    df = df[needed].copy()
    df = df[df["Fecha FIN cos"].notna()].copy()

    df = df.rename(columns={
        "Fecha Inc cos": "Fecha_Inicio_Cosecha (Primer Tallo)",
        "Fecha FIN cos": "Fecha_Fin_Cosecha",
        "Bloque": "Bloques",
        "fecha se s/p": "Fecha S/P",
        "p/s": "S/P",
        "TALLOS TOTALES EN VERDE": "Tallos_Proy",
        "AREA_PRODUCT": "Area"
    })

    df["Estado"] = "CERRADO"
    df["Variedad"] = "XL"
    df["Area"] = reemplazos_area(df["Area"])
    df["Bloques"] = df["Bloques"].astype(str)

    df = normalize_date_col(df, "Fecha S/P")
    df = normalize_date_col(df, "Fecha_Inicio_Cosecha (Primer Tallo)")
    df = normalize_date_col(df, "Fecha_Fin_Cosecha")
    df["Tallos_Proy"] = to_numeric_safe(df["Tallos_Proy"])

    return df[[
        "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
        "Tallos_Proy", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
    ]]


def fenograma_historia_clo(raw_indices_clo: pd.DataFrame, fecha_min_hist: pd.Timestamp) -> pd.DataFrame:
    df = _rebuild_header_like_clo(raw_indices_clo)
    if df.empty:
        return pd.DataFrame(columns=[
            "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
            "Tallos_Proy", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
        ])

    # Parse fechas
    if "Fecha de S/P" not in df.columns:
        raise ValueError("indices_clo_raw: no encuentro columna 'Fecha de S/P' tras reconstruir headers.")
    df["Fecha de S/P"] = to_datetime_safe(df["Fecha de S/P"])
    df["Fecha FIN cos"] = to_datetime_safe(df.get("Fecha FIN cos"))
    df["Fecha Inc cos"] = to_datetime_safe(df.get("Fecha Inc cos"))

    df = df[df["Fecha de S/P"].notna()].copy()
    df = df[df["Fecha de S/P"] >= fecha_min_hist].copy()

    needed = ["AREA_PRODUCT", "Fecha de S/P", "Fecha Inc cos", "Fecha FIN cos", "BLOQUE", "p/s", "TALLOS TOTAL
ES EN VERDE"]
    missing = [c for c in needed if c not in df.columns]
    if missing:
        raise ValueError(f"indices_clo_raw: faltan columnas requeridas: {missing}")

    df = df[needed].copy()
    df = df[df["Fecha FIN cos"].notna()].copy()

    df = df.rename(columns={
        "Fecha Inc cos": "Fecha_Inicio_Cosecha (Primer Tallo)",
        "Fecha FIN cos": "Fecha_Fin_Cosecha",
        "BLOQUE": "Bloques",
        "Fecha de S/P": "Fecha S/P",
        "p/s": "S/P",
        "TALLOS TOTALES EN VERDE": "Tallos_Proy",
        "AREA_PRODUCT": "Area"
    })

    df["Estado"] = "CERRADO"
    df["Variedad"] = "CLO"
    df["Area"] = reemplazos_area(df["Area"])
    df["Bloques"] = df["Bloques"].astype(str)
    df = normalize_date_col(df, "Fecha S/P")
    df = normalize_date_col(df, "Fecha_Inicio_Cosecha (Primer Tallo)")
    df = normalize_date_col(df, "Fecha_Fin_Cosecha")
    df["Tallos_Proy"] = to_numeric_safe(df["Tallos_Proy"])

    return df[[
        "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
        "Tallos_Proy", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
    ]]


# -------------------------
# 321A/321B preferencia (tu lógica)
# -------------------------
def preferir_con_letra_con_bloque_base(df: pd.DataFrame) -> pd.DataFrame:
    out = df.copy()
    out["Bloque_Base"] = out["Bloques"].astype(str).str.replace(r"\D", "", regex=True)

    grp_cols = ["Bloque_Base", "S/P", "Fecha S/P", "Variedad", "Area"]

    def filtrar_grupo(g: pd.DataFrame) -> pd.DataFrame:
        hay_con_letra = (g["Bloques"].astype(str) != g["Bloque_Base"].astype(str)).any()
        if hay_con_letra:
            return g[g["Bloques"].astype(str) != g["Bloque_Base"].astype(str)]
        return g

    out = out.groupby(grp_cols, dropna=False, group_keys=False).apply(filtrar_grupo)
    return out


# -------------------------
# Main (BRONZE -> SILVER only)
# -------------------------
def main() -> None:
    cfg = load_settings()
    hoy = pd.Timestamp(datetime.now().date())

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    fecha_min_hist = pd.to_datetime(cfg.get("sources", {}).get("fecha_min_hist", "2024-01-01"))
    horizon_days = int(cfg["pipeline"]["grid_horizon_days"])

    # 1) Leer fenograma activo desde BRONZE
    df_xlsm = pd.read_parquet(bronze_dir / "fenograma_xlsm_raw.parquet")
    df_xlsm = normalizar_columnas(df_xlsm)

    # 2) Lista de bloques (para filtrar balanza raw; solo optimización)
    bloques_candidatos = (
        df_xlsm.get("Bloques", pd.Series([], dtype=str))
            .dropna()
            .astype(str)
            .str.strip()
            .replace("", np.nan)
            .dropna()
            .unique()
            .tolist()
    )

    # 3) Leer balanza desde BRONZE y filtrar por bloques + fecha mínima
    bal = pd.read_parquet(bronze_dir / "balanza_bloque_fecha_raw.parquet")
    bal = normalizar_columnas(bal)
    if "Bloque" not in bal.columns or "Fecha" not in bal.columns:
        raise ValueError("balanza_bloque_fecha_raw.parquet debe tener columnas ['Bloque','Fecha'].")

    bal["Bloque"] = bal["Bloque"].astype(str)
    bal["Fecha"] = to_datetime_safe(bal["Fecha"])
    bal = bal[bal["Fecha"].notna()].copy()

    if bloques_candidatos:
        bal = bal[bal["Bloque"].isin([str(b) for b in bloques_candidatos])].copy()
    bal = bal[bal["Fecha"] >= fecha_min_hist].copy()

    # 4) Construir activo + históricos (históricos desde índices raw en BRONZE)
    activo = fenograma_activo(df_xlsm, bal, hoy)

    frames = [activo]

    # indices XL
    idx_xl_path = bronze_dir / "indices_xl_raw.parquet"
    if idx_xl_path.exists():
        raw_xl = pd.read_parquet(idx_xl_path)
        frames.append(fenograma_historia_xl(raw_xl, fecha_min_hist))

    # indices CLO
    idx_clo_path = bronze_dir / "indices_clo_raw.parquet"
    if idx_clo_path.exists():
        raw_clo = pd.read_parquet(idx_clo_path)
        frames.append(fenograma_historia_clo(raw_clo, fecha_min_hist))

    total = pd.concat(frames, ignore_index=True)

    total = normalize_date_col(total, "Fecha S/P")
    total = normalize_date_col(total, "Fecha_Inicio_Cosecha (Primer Tallo)")
    total = normalize_date_col(total, "Fecha_Fin_Cosecha")
    total = preferir_con_letra_con_bloque_base(total)

    # 5) Normalizar a fact_ciclo_maestro (silver)
    fact = total.copy()
    fact["Bloques"] = fact["Bloques"].astype(str)
    fact["Variedad"] = fact["Variedad"].astype(str)
    fact["S/P"] = fact["S/P"].astype(str).str.strip().str.upper()
    fact["Area"] = fact["Area"].astype(str)

    fact = fact.rename(columns={
        "Bloques": "bloque",
        "Variedad": "variedad",
        "S/P": "tipo_sp",
        "Fecha S/P": "fecha_sp",
        "Area": "area",
        "Tallos_Proy": "tallos_proy",
        "Fecha_Inicio_Cosecha (Primer Tallo)": "fecha_inicio_cosecha",
        "Fecha_Fin_Cosecha": "fecha_fin_cosecha",
        "Estado": "estado",
        "Bloque_Base": "bloque_base",
    })

    # Validaciones mínimas de contrato
    if fact["fecha_sp"].isna().any():
        raise ValueError("fact_ciclo_maestro: fecha_sp tiene nulos (revisar fuentes).")

    fact["bloque_id"] = fact["bloque"].map(make_bloque_id)
    fact["variedad_id"] = fact["variedad"].map(make_variedad_id)

    fact["ciclo_id"] = [
        make_ciclo_id(b, v, t, f)
        for b, v, t, f in fact[["bloque_id", "variedad_id", "tipo_sp", "fecha_sp"]].itertuples(index=False)
    ]

    # Deduplicación: prioriza ACTIVO sobre CERRADO
    fact["__prio"] = np.where(fact["estado"].astype(str).str.upper().eq("ACTIVO"), 1, 0)
    fact = (
        fact.sort_values(["ciclo_id", "__prio"], ascending=[True, False])
            .drop_duplicates(subset=["ciclo_id"], keep="first")
            .drop(columns=["__prio"])
            .reset_index(drop=True)
    )

    # Validación llave
    if fact.duplicated(["ciclo_id"]).any():
        raise ValueError("fact_ciclo_maestro: ciclo_id no es único luego de dedupe (revisar reglas).")

    write_parquet(fact, silver_dir / "fact_ciclo_maestro.parquet")
```

```
    # 6) Grid (horizonte configurable)
    grid = build_grid_ciclo_fecha(fact, horizon_days=horizon_days)
    write_parquet(grid, silver_dir / "grid_ciclo_fecha.parquet")

    print(f"OK: fact_ciclo_maestro={len(fact)} filas; grid={len(grid)} filas; horizonte={horizon_days} días")


if __name__ == "__main__":
    main()

----------------------------------------------------------------
[65/65] FILE: \src\silver\build_windows_from_milestones_final.py
----------------------------------------------------------------
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import yaml

from common.io import read_parquet, write_parquet


def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)


def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()


def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg["paths"]["silver"])

    milestones_path = silver_dir / "milestones_ciclo_final.parquet"
    if not milestones_path.exists():
        raise FileNotFoundError(f"No existe: {milestones_path}. Ejecuta primero build_milestones_final.")

    m = read_parquet(milestones_path).copy()
    m["fecha"] = _norm_date(m["fecha"])

    # Pivot
    piv = (m.pivot_table(index="ciclo_id", columns="milestone_code", values="fecha", aggfunc="min")
             .reset_index())

    # columnas
    veg_start = piv.get("VEG_START")
    hs = piv.get("HARVEST_START")
    he = piv.get("HARVEST_END")
    ps = piv.get("POST_START")
    pe = piv.get("POST_END")

    # Ventanas
    rows = []

    def add(stage: str, s: pd.Series, e: pd.Series, rule: str):
        tmp = pd.DataFrame({"ciclo_id": piv["ciclo_id"], "stage": stage})
        tmp["start_date"] = s
        tmp["end_date"] = e
        tmp["rule"] = rule
        rows.append(tmp)

    # VEG: veg_start -> hs-1 (si hs existe)
    veg_end = hs - pd.to_timedelta(1, unit="D") if hs is not None else pd.Series([pd.NaT] * len(piv))
    veg_end = veg_end.where(hs.notna(), pd.NaT) if hs is not None else veg_end
    add("VEG", veg_start, veg_end, "VEG_START -> HARVEST_START-1")

    # HARVEST: hs -> he (si hs existe)
    add("HARVEST", hs, he, "HARVEST_START -> HARVEST_END")
    # POST: ps -> pe (si ps existe)
    add("POST", ps, pe, "POST_START -> POST_END")

    win = pd.concat(rows, ignore_index=True)
    win["start_date"] = _norm_date(win["start_date"])
    win["end_date"] = _norm_date(win["end_date"])

    win = win[win["start_date"].notna()].copy()
    win["created_at"] = datetime.now().isoformat(timespec="seconds")

    out_path = silver_dir / "milestone_window_ciclo_final.parquet"
    write_parquet(win, out_path)

    print(f"OK: milestone_window_ciclo_final={len(win)} filas -> {out_path}")
    print("Stage counts:\n", win["stage"].value_counts().to_string())


if __name__ == "__main__":
    main()
```