☑ novartis_datathon_2025-Arman – Improvement TODO for Copilot Agent

> You are Copilot in VS Code, working inside the `novartis_datathon_2025-Arman` repo.
> Goal: Turn the project from "complex and good" into "sharp, reliable, and leaderboard-dangerous."

---

1. Config, Metrics & Validation Plumbing

1.1 Make Config the Single Source of Truth

- [ ] Find all places where scenario, model, sample weights, or feature choices are **hard-coded** in Python (especially in `train.py`, `inference.py`, `models/*.py`).
- [ ] Move those settings into YAML (e.g. `configs/run_defaults.yaml`, `configs/model_*.yaml`).
- [ ] Ensure all entry points load config via a shared function (e.g. `load_run_config()`), and do **not** silently override YAML settings in code.
- [ ] Add comments in YAML explaining each important flag (scenario, sample weights, model type).

1.2 Centralize Official Metric Usage

- [ ] Create or clean up a single module, e.g. `src/metrics/official.py`, that:
    - [ ] Wraps `compute_metric1` (Scenario 1) and `compute_metric2` (Scenario 2).
    - [ ] Provides helper functions:
        - [ ] `evaluate_on_train(df_train, df_pred_s1, df_pred_s2, df_aux)`
        - [ ] `evaluate_cv_fold(...)` (if needed).
- [ ] Ensure **every training script** calls these wrappers for final scoring (no custom MAE/RMSE-only selection without also logging the official metrics).
- [ ] Add a simple unit test in `tests/` that:
    - [ ] Builds tiny fake `df_actual`, `df_pred`, `df_aux`.
    - [ ] Calls `compute_metric1`, `compute_metric2`.
    - [ ] Asserts basic properties (e.g., perfect prediction → metric = 0).

1.3 Standardize Cross-Validation Strategy

- [ ] In `src/validation.py`, confirm there is a single function, e.g. `create_group_kfold_by_brand(bucket_stratified=True)`.
- [ ] Ensure **all models** (CatBoost, LGBM, XGB, etc.) use that function to get splits.
- [ ] Add a test that verifies:
    - [ ] All months of a given brand go to the same fold.
    - [ ] Bucket 1 and Bucket 2 are reasonably balanced across folds.

---

2. Strong & Stable Single-Model Baselines

2.1 Re-run Core Models on a Shared Setup

- [ ] For **Scenario 1**, run:

  - [ ] CatBoost
  - [ ] LightGBM
  - [ ] XGBoost
  - [ ] Linear / Ridge / ElasticNet (if implemented)
  - [ ] Hybrid physics-ML
  - [ ] ARIHOW
- [ ] For **Scenario 2**, run the same family of models.
- [ ] Ensure all runs:
  - [ ] Use the same CV split function.
  - [ ] Use the same feature config for that scenario.
  - [ ] Log official metrics via the centralized metric module.

2.2 Build a Model Leaderboard

- [ ] Parse `training_all_models.log` and any per-run JSON/CSV metrics to create:
  - [ ] `artifacts/model_leaderboard.csv` with columns:
    - [ ] `scenario`, `model_name`, `seed`, `cv_metric_s1`, `cv_metric_s2`, `config_path`, `timestamp`.
- [ ] Add a small script (e.g. `tools/aggregate_model_scores.py`) that regenerates the leaderboard from logs.
- [ ] Sort models by the relevant scenario metric and mark **top N** models for each scenario.

2.3 Declare “Hero” Models per Scenario

- [ ] In `docs/planning/approach.md`, document:
  - [ ] The **best single model** for Scenario 1.
  - [ ] The **best single model** for Scenario 2.
- [ ] Record:
  - [ ] Config file used.
  - [ ] Official metrics.
  - [ ] Notes on stability (e.g., variance across folds / seeds).

---

3. Time & Bucket Weighting Improvements

3.1 Review and Refine Sample Weights

- [ ] Open the sample weight section in `run_defaults.yaml` (or relevant config).
- [ ] Confirm current behavior:
  - [ ] Months 0–5, 6–11, 12–23 have different weights.
  - [ ] Bucket 1 rows have higher weight than Bucket 2 (e.g. ×2).
- [ ] Propose a **small set** of alternative weighting schemes, such as:
  - [ ] Slightly higher weight on months 0–5 (Scenario 1) and 6–11 (both scenarios).
  - [ ] Slightly higher weight on Bucket 1 (but not extreme).

3.2 Controlled Weighting Experiments

- [ ] For each candidate weight scheme:
  - [ ] Run the **hero** model for Scenario 1 and Scenario 2 with the modified weights.

    - [ ] Log official metrics and store configs separately (e.g. `configs/experiments/weights_v1.yaml`).
- [ ] Compare metrics to the baseline scheme and keep the best-performing version.
- [ ] Update `run_defaults.yaml` only if the improvement is consistent across seeds/folds.

---

4. Targeted Hyperparameter Optimization (HPO)

4.1 Define Tight Search Spaces

- [ ] For CatBoost, LGBM, and XGBoost configs:
    - [ ] Add HPO ranges for:
        - [ ] `learning_rate`
        - [ ] `depth` / `max_depth`
        - [ ] `n_estimators` (if using early stopping, allow higher max)
        - [ ] `l2_leaf_reg` / regularization parameters.
    - [ ] Keep ranges **narrow, sane, and tabular-TS-friendly**.
- [ ] Document the search spaces in comments within `model_*.yaml`.

4.2 Enable and Run Small HPO Sweeps

- [ ] Ensure `train.py` supports an HPO mode (`--hpo` or similar).
- [ ] For each scenario:
    - [ ] Run a **small** sweep for CatBoost (and optionally one GBM alternative).
    - [ ] Use the **official metric** as the optimization objective.
- [ ] Save best hyperparameters in a separate config file (e.g. `configs/model_catboost_s1_best.yaml`).

4.3 Freeze and Document Best Hyperparams

- [ ] Update the main configs to use the HPO results (or reference them).
- [ ] Keep the original defaults as commented blocks or separate "baseline" configs.
- [ ] Note in `docs/planning/approach.md` which hyperparameters came from HPO.

---

5. Smarter Ensembles

5.1 Prepare Base Models for Ensembling

- [ ] Make sure all base models used in ensembles:
    - [ ] Share the same CV splits.
    - [ ] Use the same target scaling (if any).
    - [ ] Produce predictions for the same set of brands and months.
- [ ] Add a helper in `src/models/ensemble_utils.py` to align predictions by:
    - [ ] `country`, `brand_name`, `months_postgx` keys.

5.2 Build and Evaluate Simple Ensembles

- [ ] Implement or reuse simple ensemble classes:
    - [ ] Plain average (equal weights).

  - [ ] Weighted average (weights learned on validation).
  - [ ] Simple stacking (meta-model over base predictions).
- [ ] Test ensembles like:
  - [ ] CatBoost + LightGBM.
  - [ ] CatBoost + Hybrid + ARIHOW.
- [ ] For each ensemble:
  - [ ] Compute Scenario 1 & Scenario 2 metrics.
  - [ ] Save ensemble composition and weights in JSON/YAML under `artifacts/ensembles/`.

5.3 Add Guardrails to Ensemble Predictions

- [ ] Implement a `sanitize_predictions(df_pred)` function that:
  - [ ] Clips negative volumes to 0.
  - [ ] Optionally clips very extreme values (e.g. > 5× pre-entry avg) or at least flags them.
- [ ] Call `sanitize_predictions()` before generating submission files.
- [ ] Add a small test to ensure sanitization behaves as expected.

---

6. Inference & Submission Robustness

6.1 Scenario Detection & Fallbacks

- [ ] Inspect `inference.py` logic for detecting Scenario 1 vs Scenario 2 test series.
- [ ] Add tests for:
  - [ ] Proper scenario classification given different combinations of available months.
  - [ ] Edge cases (e.g. missing some early months).
- [ ] Verify fallback strategies (e.g. exponential decay) are triggered when:
  - [ ] A model does not produce a prediction for a series/month.
  - [ ] The prediction is NaN or obviously invalid.

6.2 Submission Sanity Checks

- [ ] Implement a small script (`tools/check_submission.py`) that:
  - [ ] Reads a submission CSV.
  - [ ] Verifies:
    - [ ] Correct column names and types.
    - [ ] No negative volumes.
    - [ ] Reasonable range vs pre-entry `avg_vol` from train data.
  - [ ] Prints summary stats per scenario and bucket.
- [ ] Run this script for every candidate submission before uploading.

---

7. Experiment Tracking & Presentation Story

7.1 Experiment Tracking

- [ ] Ensure each training run:
  - [ ] Logs model name, scenario, config path, seed, official metrics to a centralized log (e.g. `training_all_models.log` + CSV).

- [ ] Add or improve a small `ExperimentTracker` utility that:
  - [ ] Appends entries to a structured file (`experiments.csv` or similar).
  - [ ] Can be queried (e.g. via a notebook) to list best runs.

7.2 Presentation Outline

- [ ] Create `docs/presentation_outline.md` with sections:
  - [ ] Problem framing (LOE, generic erosion, scenarios 1 & 2).
  - [ ] Data understanding (brands, buckets, erosion behavior).
  - [ ] Modeling approach (features, validation, models, ensembles).
  - [ ] Results and insights (official metrics, bucket-wise performance).
  - [ ] Business interpretation (why early months and Bucket 1 matter).

---

8. How Copilot Should Behave in This Repo

- [ ] Favor **small, incremental edits** over large refactors.
- [ ] Suggest moving parameters into config rather than duplicating logic.
- [ ] Auto-complete patterns based on existing code:
  - [ ] New model classes should mimic current `BaseModel` subclasses.
  - [ ] New tests follow patterns from `tests/test_smoke.py` .
- [ ] Frequently remind the user (via comments or docstrings) to:
  - [ ] Re-run `pytest` after non-trivial changes.
  - [ ] Re-run a known baseline config to check no performance regression.
  - [ ] Update docs and experiment tracker when a new “hero” result is found.

End goal:
A **clean, well-documented, and reproducible** Arman project with:

- Strong baselines per scenario,
- Thoughtful weighting and HPO,
- Smart ensembles,
- Robust inference/submission code,
- And a clear evidence trail for why your final submissions deserve to top the leaderboard.