☑ Bayesian Leaderboard Booster – TODO for Copilot Agent

> You are Copilot in VS Code, inside `novartis_datathon_2025-Arman`.
> Goal: Implement **Bayesian stacking / Bayesian model averaging** on top of existing models to improve the final leaderboard score, using the official metric structure.

---

0. Overview – What You're Building

We already have several trained base models (CatBoost, LGBM, XGB, Hybrid, ARIHOW, etc.).
We want to:

1. Collect **out-of-fold (OOF) predictions** from each base model.
2. Build a **meta-dataset** where each row is a (brand, month) and each column is a model prediction.
3. Fit a **Bayesian ensemble** that learns weights for each model, shaped by:
   - The official metric's **time-window weights** (early vs mid vs late months).
   - The **bucket weights** (Bucket 1 vs Bucket 2).
4. Apply those Bayesian ensemble weights to **test predictions** to get better submissions.

Optionally, we can also add a **hierarchical Bayesian decay model** as an extra base model.

---

1. Prerequisites & Setup

- [ ] Ensure that:
  - [ ] All base models you want to stack can:
    - [ ] Train with GroupKFold by brand.
    - [ ] Save OOF predictions for each fold.
  - [ ] You have a way to read:
    - [ ] `df_train` (with columns: `country`, `brand_name`, `months_postgx`, `volume`, `bucket`, etc.).
    - [ ] `df_aux` with `avg_vol` and `bucket` per brand (for the official metric).
- [ ] Create a new module for stacking, e.g.:
  - [ ] `src/stacking/bayesian_stacking.py`
  - [ ] Add `__init__.py` so this becomes importable.
- [ ] Optional but recommended: choose a Bayesian library:
  - [ ] PyMC (v4+),
  - [ ] or NumPyro,
  - [ ] or just use **MAP optimization** with `scipy.optimize` and treat the Dirichlet prior as a penalty.

---

2. Generate Out-of-Fold Predictions for Each Base Model

We do this **per scenario**, but the pattern is the same.

2.1 Define Base Models to Stack

- [ ] In a config file (e.g. `configs/stacking.yaml`), list base models for each scenario, for example:

```
stacking:
  scenario1_models:
    - catboost_s1
    - lgbm_s1
    - xgb_s1
    - hybrid_s1
    - arihow_s1
  scenario2_models:
    - catboost_s2
    - lgbm_s2
    - xgb_s2
    - hybrid_s2
    - arihow_s2
```

- [ ] Ensure each model name maps to a config / training function.

2.2 Implement OOF Prediction Generation

- [ ] In `src/stacking/bayesian_stacking.py`, implement a helper:

```
def generate_oof_predictions(model_name: str, scenario: int, config) ->
pd.DataFrame:
    """
    Trains model with GroupKFold by brand and returns OOF predictions
DataFrame with:
    columns = [country, brand_name, months_postgx, y_true, y_pred, bucket]
    for the specified scenario.
    """
```

Key details:

- [ ] Use the **same CV splits** used in your normal training (GroupKFold by brand, bucket-stratified).
- [ ] For each fold:
  - [ ] Train the model on train folds.
  - [ ] Predict on validation fold.
  - [ ] Store predictions along with `y_true`, brand info, and `months_postgx`.

- [ ] Concatenate all folds to form a full OOF DataFrame.
- [ ] Restrict months based on scenario:
  - [ ] Scenario 1: 0–23.
  - [ ] Scenario 2: 6–23.
- [ ] Save the OOF predictions to disk:
  - [ ] `artifacts/oof/{model_name}_scenario{scenario}.parquet`.

2.3 Run OOF Generation for All Base Models

- [ ] Write a small script, e.g. `tools/run_oof_for_stacking.py`, that:

```
for scenario in [1, 2]:
    for model_name in config.stacking[f"scenario{scenario}_models"]:
        generate_oof_predictions(model_name, scenario, config)
```

- [ ] Run this script once to populate OOF files.

---

3. Build the Meta-Dataset for Stacking

We now combine all base-model OOF predictions into a **single meta-table** per scenario.

3.1 Load and Join OOF Predictions

- [ ] Implement:

```
def build_meta_dataset_for_scenario(scenario: int, config) -> pd.DataFrame:
    """
    Returns a DataFrame where each row = (country, brand_name, months_postgx),
    columns = [y_true, bucket, base_model_1, base_model_2, ..., base_model_M].
    """
```

Steps:

- [ ] For the scenario, load all OOF files listed in `config.stacking`.
- [ ] Start from the first model's OOF DataFrame as the base.
- [ ] Sequentially merge other models' OOFs on keys:
  - [ ] `country`
  - [ ] `brand_name`
  - [ ] `months_postgx`
- [ ] Ensure `y_true` and `bucket` are consistent across joins.
- [ ] Resulting DataFrame columns example:

```
country, brand_name, months_postgx, y_true, bucket,
pred_catboost, pred_lgbm, pred_xgb, pred_hybrid, pred_arihow
```

- [ ] Save this meta-dataset as:
  - [ ] `artifacts/stacking/meta_scenario{scenario}.parquet`.

3.2 Add Metric-Based Sample Weights

Recall the official metric gives more importance to:

- Early months (0–5 in S1, 6–11 in both),
- Bucket 1 (weighted 2x vs Bucket 2).
- [ ] In `build_meta_dataset_for_scenario`, add a column `sample_weight`:

```python
def compute_sample_weight(month, bucket, scenario):
    # Time-window weights (example values, tune if needed)
    if scenario == 1:
        if 0 <= month <= 5:
            time_w = 3.0
        elif 6 <= month <= 11:
            time_w = 4.0
        else:
            time_w = 2.0
    elif scenario == 2:
        if 6 <= month <= 11:
            time_w = 4.0
        else:
            time_w = 2.0

    bucket_w = 2.0 if bucket == 1 else 1.0
    return time_w * bucket_w
```

- [ ] Apply this to each row and store as `sample_weight`.

This shapes stacking to care most about the same regions the competition metric cares about.

---

4. Implement Bayesian Stacking

We want a **convex combination** of base model predictions with **Dirichlet prior** over weights.

4.1 Model Definition (Conceptual)

For each row (n) in the meta-dataset (brand + month):

- Base preds: (x_{n,1}, x_{n,2}, ..., x_{n,M}) (M models).
- Ensemble prediction:

[\hat{y}*n = \sum*{m=1}^M w_m x_{n,m}]

- Weights:

[\mathbf{w} \sim \text{Dirichlet}(\alpha_1,...,\alpha_M)]

- Likelihood (with sample weights (s_n)):

[y_n \sim \mathcal{N}(\hat{y}_n, \sigma^2)]

and we treat the loss as:

[L(\mathbf{w}) = \sum_n s_n (y_n - \hat{y}_n)^2 - \log p(\mathbf{w})]

4.2 MAP Optimization Implementation (Practical)

Instead of full MCMC (which may be heavy), we can do **MAP optimization** using scipy.

- ☐ Implement in `bayesian_stacking.py`:

```
import numpy as np
from scipy.optimize import minimize

def fit_dirichlet_weighted_ensemble(X, y, sample_weight, alpha=None):
    """
    X: shape (N, M) matrix of base predictions
    y: shape (N,) true targets
    sample_weight: shape (N,) sample weights (time * bucket)
    alpha: shape (M,) Dirichlet prior params (default: all ones)
    Returns: weights w (shape (M,))
    """
```

Implementation details:

- ☐ If `alpha` is None, set `alpha = np.ones(M)`.
- ☐ Reparameterize weights with softmax for positivity and sum-to-one:

```
def softmax(z):
    e = np.exp(z - np.max(z))
    return e / e.sum()
```

- ☐ Define the loss function on unconstrained params `z`:

```
def loss_fn(z):
    w = softmax(z)  # shape (M,)
    y_pred = X @ w  # (N,)
    resid = y - y_pred
    # weighted squared error
    mse_part = np.sum(sample_weight * resid**2)
    # negative log Dirichlet prior (up to const)
    # p(w) ∝ ∏ w_m^(alpha_m - 1)
    # -log p(w) = -∑ (alpha_m - 1) log w_m
    prior_part = -np.sum((alpha - 1.0) * np.log(w + 1e-12))
    return mse_part + prior_part
```

- [ ] Initialize `z = np.zeros(M)` (uniform weights).
- [ ] Call `minimize(loss_fn, z, method="L-BFGS-B")`.
- [ ] Convert optimal `z` back to `w = softmax(z_opt)`.
- [ ] Return `w`.
- [ ] Wrap this as a class:

```
class BayesianStacker:
    def __init__(self, alpha=None):
        self.alpha = alpha
        self.weights_ = None

    def fit(self, X, y, sample_weight):
        # store M
        # run fit_dirichlet_weighted_ensemble
        # save self.weights_
        return self

    def predict(self, X):
        return X @ self.weights_
```

4.3 Fit Stacker per Scenario

- [ ] Implement:

```
def train_stacker_for_scenario(scenario: int, config):
    df_meta = load_meta_dataset_for_scenario(scenario)
    base_cols = [c for c in df_meta.columns if c.startswith("pred_")]
    X = df_meta[base_cols].to_numpy()
    y = df_meta["y_true"].to_numpy()
```

```
    sample_weight = df_meta["sample_weight"].to_numpy()

    stacker = BayesianStacker(alpha=np.ones(len(base_cols)))
    stacker.fit(X, y, sample_weight)

    # Save weights and metadata
    np.save(f"artifacts/stacking/weights_scenario{scenario}.npy",
stacker.weights_)
    with open(f"artifacts/stacking/weights_scenario{scenario}.txt", "w") as f:
        for name, w in zip(base_cols, stacker.weights_):
            f.write(f"{name}: {w:.4f}\n")
```

- [ ] Run this for Scenario 1 and Scenario 2.

---

5. Apply Bayesian Ensemble to Test Predictions

5.1 Collect Base Test Predictions

- [ ] For each base model and scenario, ensure you can load test predictions:

```
# Example expected columns: country, brand_name, months_postgx, volume_pred
df_test_model =
pd.read_parquet(f"artifacts/test_preds/{model_name}_scenario{scenario}.parquet
")
```

- [ ] Similar to the meta-dataset, merge all model predictions on:
  - `country`, `brand_name`, `months_postgx`.
- [ ] Build a test matrix `X_test` with same order of base model columns as in training:
  - `pred_catboost`, `pred_lgbm`, ..., `pred_arihow`.

5.2 Apply Learned Weights

- [ ] Load ensemble weights:

```
weights = np.load(f"artifacts/stacking/weights_scenario{scenario}.npy")
```

- [ ] Compute ensemble predictions:

```
y_ens = X_test @ weights
df_test_ens = df_base[["country", "brand_name", "months_postgx"]].copy()
```

```
    df_test_ens["volume"] = y_ens
```

- [ ] Apply any **sanitization** (e.g., clip negatives to 0).

5.3 Export Final Submission Files

- [ ] For Scenario 1:
  - [ ] Ensure `months_postgx` = 0–23 for each test brand.
  - [ ] Save CSV: `submissions/ensemble_scenario1.csv` with columns `country,brand_name,months_postgx,volume`.
- [ ] For Scenario 2:
  - [ ] Ensure `months_postgx` = 6–23 for each test brand.
  - [ ] Save CSV: `submissions/ensemble_scenario2.csv` with the same columns.
- [ ] Run your existing **submission checker** (if any) to validate shapes and ranges.

---

6. Optional: Full Bayesian Sampling & Submission Diversification

If you have time and want extra leaderboard robustness:

- [ ] Instead of just MAP, sample **multiple** sets of weights `w^(k)` by:
  - [ ] Running MAP from different random initializations,
  - [ ] Or adding small Gaussian noise around the MAP solution,
  - [ ] Or using a light-weight MCMC over `z` (unconstrained params).
- [ ] For each `w^(k)`:
  - [ ] Generate a slightly different ensemble prediction.
  - [ ] Save as a different submission pair (Scenario 1 and 2).

This gives you **legit diversified submissions** that all respect the physics and the metric, but hedge over uncertainty in ensemble weights.

---

7. Optional: Hierarchical Bayesian Erosion Model as Extra Base Model

If you want a more "pure" Bayesian model in the stack:

- [ ] Implement a simple hierarchical model:
  - [ ] For each brand j:
    - [ ] Fit `vol_norm_{j}(t) ≈ a_j * exp(-b_j t) + c_j`.
  - [ ] Put priors on `a_j, b_j, c_j` by therapeutic area / bucket (hierarchical priors).
- [ ] Fit using PyMC/Stan/NumPyro on the **normalized volumes**.
- [ ] Generate predictive mean for months 0–23 / 6–23.
- [ ] Treat this model as an additional base column `pred_bayes_decay` in the meta-dataset.

- [ ] Re-run the Bayesian stacking with the new base model included.

This often helps because the hierarchical decay model encodes “physics” that GBMs miss.

---

8. Testing & Sanity Checks

- [ ] Add tests for `BayesianStacker`:
  - [ ] On synthetic data, check that if one model is always better, the stacker gives it higher weight.
  - [ ] Check that weights are non-negative and sum to 1.
- [ ] Validate meta-data merges:
  - [ ] Ensure no rows are dropped when joining OOFs.
  - [ ] Confirm that `y_true` is identical across models for the same (country, brand, month).
- [ ] Compare:
  - [ ] Official metrics of the best single model vs the Bayesian ensemble.
  - [ ] Ensure the ensemble is not worse; otherwise debug weighting / data alignment.

End Goal:
You have a **Bayesian ensemble layer** that:

- Uses all your existing models intelligently,
- Respects the competition’s weird metric (time & bucket weights),
- Produces more robust, higher-scoring submissions,
- And can generate multiple “Bayesian-flavored” submission variants to hedge risk on the final leaderboard.