Bayesian Stacking Implementation Guide

> Complete guide for the Bayesian stacking, hierarchical decay model, and submission diversification features implemented in the `novartis_datathon_2025-Arman` project.

Table of Contents

Overview

Goal

The Bayesian stacking system improves leaderboard scores by:

1. **Intelligently combining multiple models** using learned weights instead of simple averaging
2. **Respecting the competition metric structure** by weighting samples according to time windows and buckets
3. **Adding physics-informed predictions** via a hierarchical Bayesian decay model
4. **Hedging leaderboard risk** through diversified submission variants

How It Works

```
Base Models (CatBoost, LightGBM, XGBoost, Hybrid, etc.)
                    ↓
         Out-of-Fold Predictions
                    ↓
              Meta-Dataset
                    ↓
    Bayesian Stacker (Dirichlet Prior + MAP)
                    ↓
         Learned Ensemble Weights
                    ↓
    Apply to Test → Multiple Submissions
```

Architecture

Files Created/Modified

```
src/stacking/
├── __init__.py                    # Module exports
└── bayesian_stacking.py           # Main implementation (~1200 lines)

tools/
├── run_oof_for_stacking.py        # Generate OOF predictions
├── train_bayesian_stacker.py      # Train the Bayesian stacker
├── generate_ensemble_submission.py # Generate ensemble submissions
├── run_hierarchical_decay.py      # Run hierarchical decay model
└── generate_diversified_submissions.py  # Generate diversified submissions

configs/
└── stacking.yaml                  # Stacking configuration

tests/
└── test_bayesian_stacking.py      # Unit tests (24 tests)
```

Installation & Setup

Prerequisites

```
# Ensure you're in the project directory
cd D:\Datathon\novartis_datathon_2025\novartis_datathon_2025-Arman

# Activate virtual environment
.\.venv\Scripts\Activate

# Install dependencies (if not already installed)
pip install -r requirements.txt
```

Required Dependencies

The following are used (already in requirements.txt):

- `numpy`
- `pandas`
- `scipy` (for optimization)

- `scikit-learn` (for GroupKFold)
- `joblib` (for model persistence)
- `pyyaml` (for config loading)

Core Components

1. BayesianStacker

Goal: Learn optimal convex combination weights for base model predictions using MAP optimization with a Dirichlet prior.

Why Dirichlet Prior?

- Ensures weights are positive and sum to 1 (valid probability simplex)
- Provides regularization toward uniform weights (prevents overfitting)
- Interpretable: higher alpha = stronger prior toward uniform

Mathematical Formulation:

For each sample n with base predictions $x_{n,1}, ..., x_{n,M}$:

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

Weights have Dirichlet prior: $$\mathbf{w} \sim \text{Dirichlet}(\alpha_1, ..., \alpha_M)$$

Loss function (MAP objective): $$L(\mathbf{w}) = \sum_n s_n (y_n - \hat{y}_n)^2 - \sum_m (\alpha_m - 1) \log w_m$$

Where s_n is the sample weight based on time window and bucket.

Sample Weights:

The official metric emphasizes certain regions more:

Scenario	Months 0-5	Months 6-11	Months 12-23
S1	3.0	4.0	2.0
S2	N/A	4.0	2.0

Bucket multiplier: Bucket 1 = 2x, Bucket 2 = 1x

Code Example:

```
from src.stacking import BayesianStacker

# Create stacker
stacker = BayesianStacker(
    alpha=np.ones(5),           # Uniform Dirichlet prior
    regularization_strength=1.0,
    clip_predictions=True,
```

```
        clip_min=0.0,
        clip_max=2.0
    )

    # Fit on OOF predictions
    stacker.fit(X_oof, y_true, sample_weight, model_names=['cat', 'lgbm', 'xgb',
    'hybrid', 'arihow'])

    # Get learned weights
    print(stacker.get_weights_dict())
    # {'cat': 0.35, 'lgbm': 0.25, 'xgb': 0.20, 'hybrid': 0.15, 'arihow': 0.05}

    # Apply to test
    y_ensemble = stacker.predict(X_test)
```

---

2. Hierarchical Bayesian Decay Model

Goal: Add a physics-informed base model that captures the exponential erosion pattern of generic drug entry.

Why This Helps:

- GBM models may miss the underlying decay dynamics
- Encodes domain knowledge: volume erodes exponentially after generic entry
- Hierarchical priors allow borrowing strength across similar brands

Decay Model:

For brand j at time t: $$\text{vol_norm}_j(t) = a_j \cdot e^{-b_j \cdot t} + c_j$$

Where:

- a_j: Initial amplitude (starting normalized volume)
- b_j: Decay rate (higher = faster erosion)
- c_j: Asymptote (final steady-state volume)

Hierarchical Priors:

Parameters are shrunk toward bucket-level means: $$a_j \sim \mathcal{N}(\mu_a^{(bucket)}, \sigma_a^{(bucket)})$$

This allows:

- Fast-decay brands (Bucket 1) to share information
- Slow-decay brands (Bucket 2) to share information

Empirical Bayes Approach:

1. First pass: Fit all brands with global priors
2. Estimate bucket-specific prior means/stds from fitted parameters
3. Second pass: Refit with hierarchical priors + shrinkage

Code Example:

```
from src.stacking import HierarchicalBayesianDecay

# Create model with hierarchical priors
model = HierarchicalBayesianDecay(
    prior_a_mean=1.0,
    prior_a_std=0.5,
    prior_b_mean=0.05,
    prior_b_std=0.02,
    prior_c_mean=0.3,
    prior_c_std=0.2,
    use_hierarchical_priors=True,
    shrinkage_strength=0.3
)

# Fit on training data
model.fit(df_train, target_col='y_norm')

# Generate predictions
predictions = model.predict_fast(df_test)

# Save for later use
model.save('artifacts/stacking/hierarchical_decay.joblib')
```

---

3. Submission Diversification

Goal: Generate multiple legitimate submission variants to hedge risk on the private leaderboard.

Why Diversify?

- Public leaderboard may not represent private leaderboard distribution
- Different weight configurations may perform better on different data splits
- Reduces variance in final ranking

Three Methods:

Method 1: Multi-Init (Recommended)

Runs MAP optimization from multiple random initializations to find different local optima.

```
from src.stacking import fit_stacker_multi_init

# Get multiple weight configurations
all_weights = fit_stacker_multi_init(
    X_train, y_train, sample_weight,
    n_inits=10  # Try 10 different initializations
```

```
)
# Returns list of diverse weight vectors
```

Method 2: MCMC Sampling

Light-weight Metropolis-Hastings sampling from the posterior over weights.

```
from src.stacking import mcmc_sample_weights

# Sample from posterior
weight_samples = mcmc_sample_weights(
    X_train, y_train, sample_weight,
    n_samples=10,    # Number of samples to collect
    step_size=0.05,  # Random walk step size
    burn_in=50,      # Burn-in iterations
    thin=5           # Thinning interval
)
```

Method 3: Noise Perturbation

Adds Gaussian noise to MAP weights in log space.

```
from src.stacking import generate_diversified_submissions

# Generate variants using noise
submissions = generate_diversified_submissions(
    stacker, test_pred_files, scenario=1,
    n_variants=5,
    noise_std=0.1  # Noise in log-weight space
)
```

Blend of Blends:

Create a final submission by averaging all variants:

```
from src.stacking import create_blend_of_blends

# Equal weight blend
final_submission = create_blend_of_blends(submissions)

# Custom weights (e.g., weight MAP higher)
final_submission = create_blend_of_blends(
    submissions,
```

```
    weights=[0.4, 0.15, 0.15, 0.15, 0.15]
)
```

Configuration

stacking.yaml

```yaml
#
================================================================================
# BASE MODELS TO STACK
#
================================================================================
stacking:
  scenario1_models:
    - catboost_s1
    - lgbm_s1
    - xgb_s1
    - hybrid_s1

  scenario2_models:
    - catboost_s2
    - lgbm_s2
    - xgb_s2
    - hybrid_s2

#
================================================================================
# BAYESIAN STACKER SETTINGS
#
================================================================================
bayesian_stacker:
  alpha: 1.0                    # Dirichlet concentration (1.0 = uniform
prior)
  regularization_strength: 1.0  # Prior strength multiplier
  clip_predictions: true
  clip_min: 0.0
  clip_max: 2.0
  output_dir: artifacts/stacking

#
================================================================================
# SAMPLE WEIGHT SETTINGS
#
================================================================================
sample_weights:
  time:
    s1_early: 3.0   # Months 0-5 (S1 only)
    s1_mid: 4.0     # Months 6-11
    s1_late: 2.0    # Months 12-23
```

```yaml
    s2_mid: 4.0     # Months 6-11
    s2_late: 2.0    # Months 12-23
  bucket:
    bucket_1: 2.0   # Fast decay
    bucket_2: 1.0   # Slow decay

#
# ============================================================================
# HIERARCHICAL BAYESIAN DECAY (Section 7)
#
# ============================================================================
hierarchical_decay:
  enabled: true
  use_hierarchical_priors: true
  shrinkage_strength: 0.3
  prior:
    a_mean: 1.0
    a_std: 0.5
    b_mean: 0.05
    b_std: 0.02
    c_mean: 0.3
    c_std: 0.2

#
# ============================================================================
# SUBMISSION DIVERSIFICATION (Section 6)
#
# ============================================================================
diversification:
  enabled: true
  method: multi_init  # Options: 'multi_init', 'mcmc', 'noise'
  n_variants: 5
  noise_std: 0.1      # For 'noise' method
  mcmc:
    step_size: 0.05
    burn_in: 50
    thin: 5
  create_blend: true
  output_dir: submissions/ensemble
```

Usage Commands

Complete Workflow

```powershell
# Navigate to project directory
cd D:\Datathon\novartis_datathon_2025\novartis_datathon_2025-Arman

# Activate environment
.\.venv\Scripts\Activate
```

Step 1: Generate OOF Predictions

Generate out-of-fold predictions for all base models.

```
python tools/run_oof_for_stacking.py --scenario 1 2
```

What it does:

- Trains each model with GroupKFold by brand
- Saves OOF predictions to `artifacts/oof/{model}_scenario{s}.parquet`
- Uses same CV splits across all models for fair comparison

Output:

```
artifacts/oof/
├── catboost_s1_scenario1.parquet
├── lgbm_s1_scenario1.parquet
├── xgb_s1_scenario1.parquet
├── hybrid_s1_scenario1.parquet
├── catboost_s2_scenario2.parquet
└── ...
```

Step 2: Run Hierarchical Decay Model (Optional but Recommended)

Add the physics-informed decay model as an extra base.

```
python tools/run_hierarchical_decay.py --scenario 1 2
```

What it does:

- Fits hierarchical decay model with Empirical Bayes priors
- Generates OOF predictions for the decay model
- Adds `pred_bayes_decay` column to meta-dataset
- Re-trains stacker with enhanced meta-dataset

Output:

```
artifacts/oof/bayes_decay_scenario1.parquet
artifacts/stacking/hierarchical_decay_scenario1.joblib
```

```
artifacts/stacking/meta/meta_scenario1_with_decay.parquet
```

Step 3: Train Bayesian Stacker

Fit the Bayesian stacker on the meta-dataset.

```
python tools/train_bayesian_stacker.py --scenario 1 2
```

What it does:

- Loads all OOF predictions
- Builds meta-dataset with sample weights
- Fits BayesianStacker using MAP optimization
- Saves learned weights

Output:

```
artifacts/stacking/
├── stacker_scenario1.joblib
├── weights_scenario1.npy
├── weights_scenario1.txt
├── stacker_scenario2.joblib
├── weights_scenario2.npy
└── weights_scenario2.txt
```

Example weights_scenario1.txt:

```
Scenario 1 Bayesian Stacking Weights
==================================================
catboost: 0.312456
lgbm: 0.256789
xgb: 0.198234
hybrid: 0.145678
bayes_decay: 0.086843
```

Step 4: Generate Ensemble Submission

Create the main ensemble submission.

```
python tools/generate_ensemble_submission.py --scenario 1 2
```

Output:

```
submissions/
├── ensemble_scenario1.csv
└── ensemble_scenario2.csv
```

Step 5: Generate Diversified Submissions

Create multiple submission variants for leaderboard hedging.

```
# Using multi-init (recommended)
python tools/generate_diversified_submissions.py --scenario 1 2 --method
multi_init

# Using MCMC sampling
python tools/generate_diversified_submissions.py --scenario 1 2 --method mcmc

# Using noise perturbation
python tools/generate_diversified_submissions.py --scenario 1 2 --method noise

# With custom number of variants
python tools/generate_diversified_submissions.py --scenario 1 2 --method
multi_init --n-variants 10
```

Output:

```
submissions/ensemble/
├── ensemble_scenario1_map.csv           # Original MAP solution
├── ensemble_scenario1_multi_init_1.csv  # Variant 1
├── ensemble_scenario1_multi_init_2.csv  # Variant 2
├── ensemble_scenario1_multi_init_3.csv  # Variant 3
├── ensemble_scenario1_multi_init_4.csv  # Variant 4
├── ensemble_scenario1_blend.csv         # Blend of all variants
├── ensemble_scenario2_map.csv
├── ensemble_scenario2_multi_init_1.csv
└── ...
```

API Reference

BayesianStacker

```python
class BayesianStacker:
    def __init__(
        self,
        alpha: Optional[np.ndarray] = None,      # Dirichlet prior params
        regularization_strength: float = 1.0,    # Prior strength
        clip_predictions: bool = True,
        clip_min: float = 0.0,
        clip_max: float = 2.0
    )

    def fit(
        self,
        X: np.ndarray,                           # (N, M) OOF predictions
        y: np.ndarray,                           # (N,) true targets
        sample_weight: Optional[np.ndarray],     # (N,) sample weights
        model_names: Optional[List[str]]
    ) -> 'BayesianStacker'

    def predict(self, X: np.ndarray) -> np.ndarray

    def get_weights_dict(self) -> Dict[str, float]

    def save(self, path: str) -> None

    @classmethod
    def load(cls, path: str) -> 'BayesianStacker'
```

HierarchicalBayesianDecay

```python
class HierarchicalBayesianDecay:
    def __init__(
        self,
        prior_a_mean: float = 1.0,
        prior_a_std: float = 0.5,
        prior_b_mean: float = 0.05,
        prior_b_std: float = 0.02,
        prior_c_mean: float = 0.3,
        prior_c_std: float = 0.2,
        use_hierarchical_priors: bool = True,
        shrinkage_strength: float = 0.3
    )

    def fit(
        self,
```

```
        df: pd.DataFrame,
        target_col: str = 'y_norm',
        time_col: str = 'months_postgx'
    ) -> 'HierarchicalBayesianDecay'

    def predict(self, df: pd.DataFrame) -> np.ndarray

    def predict_fast(self, df: pd.DataFrame) -> np.ndarray  # Vectorized

    def save(self, path: str) -> None

    @classmethod
    def load(cls, path: str) -> 'HierarchicalBayesianDecay'
```

Utility Functions

```
# Sample weight computation
def compute_sample_weight(month: int, bucket: int, scenario: int) -> float

def compute_sample_weights_vectorized(
    months: np.ndarray,
    buckets: np.ndarray,
    scenario: int
) -> np.ndarray

# OOF generation
def generate_oof_predictions(
    model_class, model_config, df_train, feature_cols, ...
) -> pd.DataFrame

# Meta-dataset building
def build_meta_dataset_for_scenario(
    oof_files: Dict[str, str],
    scenario: int
) -> pd.DataFrame

# Diversification
def fit_stacker_multi_init(X, y, sample_weight, n_inits=10) ->
List[np.ndarray]

def mcmc_sample_weights(X, y, sample_weight, n_samples=100, ...) ->
List[np.ndarray]

def generate_bayesian_submission_variants(
    X_train, y_train, sample_weight_train,
    test_pred_files, scenario, method, n_variants, ...
) -> List[pd.DataFrame]

def create_blend_of_blends(
    submissions: List[pd.DataFrame],
```

```
        weights: Optional[np.ndarray] = None
    ) -> pd.DataFrame
```

Testing

Run All Tests

```
python -m pytest tests/test_bayesian_stacking.py -v
```

Run Specific Test Classes

```
# Test BayesianStacker
python -m pytest tests/test_bayesian_stacking.py::TestBayesianStacker -v

# Test Hierarchical Decay
python -m pytest
tests/test_bayesian_stacking.py::TestHierarchicalBayesianDecay -v

# Test Diversification
python -m pytest tests/test_bayesian_stacking.py::TestDiversification -v
```

Test Coverage

Test Class	Tests	Description
TestSoftmax	3	Numerical stability, sum-to-one, positivity
TestComputeSampleWeight	4	Time/bucket weighting, vectorized version
TestFitDirichletWeightedEnsemble	4	Weight optimization, sample weight effects
TestBayesianStacker	5	Fit/predict, clipping, save/load
TestMetaDatasetBuilding	1	OOF alignment verification
TestHierarchicalBayesianDecay	3	Fit/predict, hierarchical priors, save/load
TestDiversification	4	Multi-init, MCMC, blending

Leaderboard Strategy

Recommended Submission Strategy

1. **Primary Submission:** `ensemble_scenario{1,2}_map.csv`

- The MAP solution with optimal weights
- Usually the best single submission

2. **Backup Submission:** `ensemble_scenario{1,2}_blend.csv`
   - Blend of all variants
   - Most stable, lower variance
3. **Hedge Submissions:** `ensemble_scenario{1,2}_multi_init_{1-4}.csv`
   - Different local optima
   - May outperform on private leaderboard

When to Use Each Method

Method	When to Use	Pros	Cons
multi_init	Default choice	Fast, finds diverse optima	May miss global optimum
mcmc	Want Bayesian uncertainty	Samples from true posterior	Slower, needs tuning
noise	Quick diversification	Very fast	Less principled

Tuning Tips

1. **If stacker overfits:**
   - Increase `regularization_strength` (try 2.0, 5.0)
   - Increase Dirichlet `alpha` (try 2.0, 5.0)
2. **If single model dominates:**
   - This is fine! It means that model is genuinely best
   - The stacker correctly identifies it
3. **If hierarchical decay hurts performance:**
   - Set `hierarchical_decay.enabled: false`
   - Or reduce `shrinkage_strength`
4. **For more diverse submissions:**
   - Increase `n_variants` (try 10)
   - Use different methods and combine

Troubleshooting

Common Issues

Issue: "Meta-dataset not found"

```
Solution: Run train_bayesian_stacker.py first to create the meta-dataset
```

Issue: "Test predictions not found"

```
Solution: Run base model predictions first using the normal training pipeline
```

Issue: "MCMC acceptance rate too low"

```
Solution: Reduce step_size in config (try 0.02)
```

Issue: "Weights are nearly uniform"

```
This may be correct if all models perform similarly.
Check individual model scores to verify.
```

Summary

The Bayesian stacking system provides:

1. ☑ **Intelligent Model Combination** - Learns optimal weights using MAP with Dirichlet prior
2. ☑ **Metric-Aware Weighting** - Respects official competition metric structure
3. ☑ **Physics-Informed Predictions** - Hierarchical decay model captures erosion dynamics
4. ☑ **Leaderboard Hedging** - Multiple diversification methods for robust submissions
5. ☑ **Production-Ready** - CLI tools, configuration, tests, and documentation

Total Implementation:

- ~1,200 lines in `bayesian_stacking.py`
- 5 CLI tools
- 24 unit tests
- Comprehensive configuration