Slow21 Survival AFT Model with Stock-Augmented Features

A technical thesis-style reconstruction of the training pipeline, statistical model, and tree-building algorithms

Generated: 2026-01-14

Scope: iPhone listing time-to-sell modeling (training script: *train_slow21_gate_classifier_pg.py*, with optional Python-only stock features).

Key deliverable. This document explains, in detail, how the model is trained, how the boosted trees are constructed, how censoring is handled under the Accelerated Failure Time (AFT) objective, and how the three stock features are computed in Python and audited for leakage risk.

Table of Contents

Abstract

We model time-to-sell for iPhone marketplace listings using a censored survival objective under an Accelerated Failure Time (AFT) formulation, implemented via XGBoost's *survival:aft* objective with histogram-based, leaf-wise tree growth (*grow_policy=lossguide*). The pipeline joins multiple *t0-safe* feature stores (socio/market, geo, vision, fusion, device meta, trainer-derived) and optionally adds three Python-only stock features capturing contemporaneous live inventory within (super_metro, generation) segments. We detail the statistical model, censored likelihood, gradient-boosted tree construction, weighting and calibration strategy, and explainability layer (feature-block SHAP attribution). We then summarize exploratory evidence that stock features provide conditional signal and propose a rigorous ablation protocol to verify out-of-sample value without leakage.

Three stock features (Python-only):

• **stock_n_sm4_gen_sbucket**

• **stock_n_sm4_gen**

• **stock_share_sbucket** = stock_n_sm4_gen_sbucket / NULLIF(stock_n_sm4_gen, 0)

Primary model outputs. The model is evaluated as (i) a time-to-sell regressor in hours and (ii) a Slow21 classifier using the threshold 504 hours (=21 days).

1. Problem Formulation

Given a marketplace listing observed at *t0* (the feature extraction timestamp, typically the listing's edited_date), we seek to predict the listing's time-to-sell T (in hours) and—operationally—whether the listing is likely to be "slow" as defined by the Slow21 gate:

$$p_{\text{slow21}}(\mathbf{x}) = \mathbb{P}(T \geq 504 \text{ h} \mid \mathbf{x}) = S(504 \text{ h} \mid \mathbf{x})$$

Equation 1. Slow21 classification target as a survival probability.

This problem is inherently a survival-analysis setting because many listings are not observed to sell within the observation window, creating right-censoring. A survival formulation allows us to use both sold and censored examples coherently, rather than discarding censored observations.

The training script implements this via an AFT (Accelerated Failure Time) objective trained with gradient-boosted decision trees.

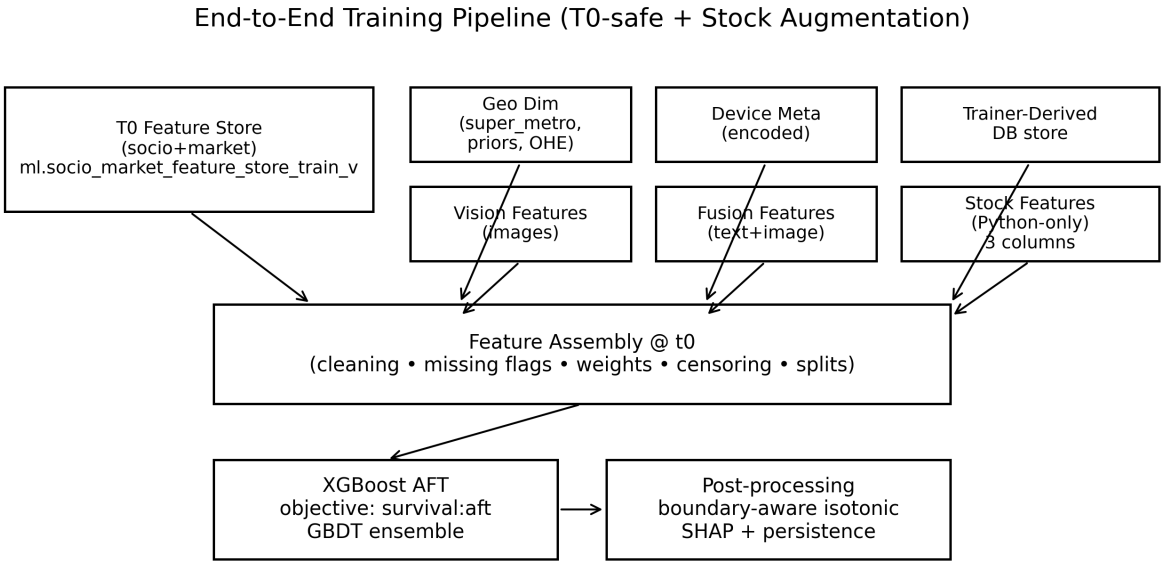

Figure 1. End-to-end training pipeline (feature assembly, AFT training, calibration, explainability).

2. Data, Time Indexing, and Leakage Constraints

The pipeline is built around a strict **t0 snapshot principle**: every predictive feature must be computable using information available at or before the listing's t0. In the training script, t0 is represented by the *edited_date* column (UTC-normalized).

This constraint is essential because marketplace data are naturally longitudinal (listings evolve over time). If features inadvertently incorporate post-t0 behavior (e.g., last_seen, sold_date, subsequent price cuts), the model will show inflated offline metrics and fail in production (classic leakage).

2.1 Primary data sources (feature stores)

The training dataset is produced by joining a base listing table with multiple feature-store views. Each view is expected to be t0-aligned and (optionally) certified for freshness via a max-age contract check.

Block	Typical source view (examples from CLI)	Description
socio_economic_store	ml.socio_market_feature_store_train_v	Socio/market covariates and relative pricing features
geo_store	ml.geo_dim_super_metro_v4_t0_train_v	Geo segmentation, one-hot regions, and shrinkage priors
vision_store	ml.iphone_image_features_unified_v1_train_v	Image-derived signals (e.g., photo quality, battery cues)
fusion_store	ml.v_damage_fusion_features_v2_scored_train_v	Text+image fusion features (damage, battery, etc.)
device_meta_store	ml.iphone_device_meta_encoded_v2_t0_train_v	Encoded device metadata and category signals
trainer_derived_store	ml.trainer_derived_features_v1_mv	Trainer-derived aggregates (flow, anchors, derived counts)
stock_store (Python-only)	computed at runtime	Live-inventory features computed inside Python (no DB MV/view)

A critical implementation detail is that the script also creates explicit **missingness flags** for some stores. This stabilizes the model when upstream stores are partially missing and ensures the tree splits can condition on the presence/absence of signals rather than imputed values alone.

2.2 Certification and freshness contracts

The CLI includes *--cert_max_age_hours*, which is used to assert that upstream feature-store views were recently refreshed. This is not about leakage; it is an operational guardrail ensuring that training does not silently consume stale feature snapshots.

3. Labeling, Censoring, and Evaluation Protocol

The target variable is the **time-to-sell** in hours. For sold listings, the sale time is observed and an exact duration can be computed. For listings that have not sold within the observation horizon (or that become inactive), the final sale time is unknown: those examples are treated as **right-censored**.

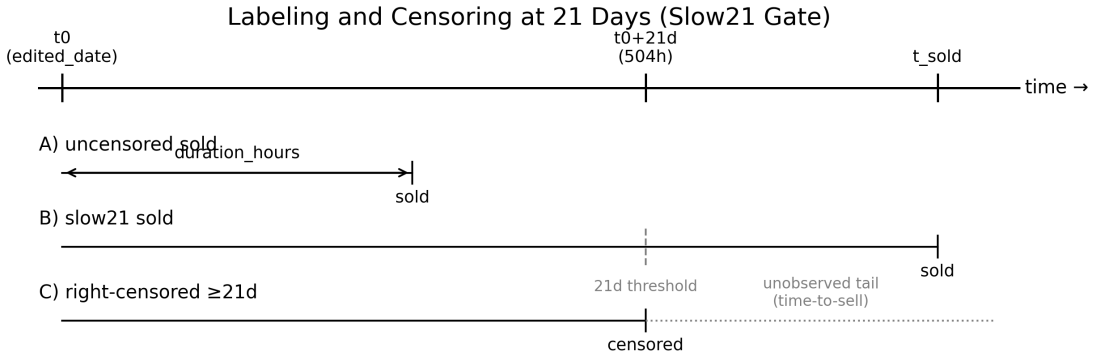

Figure 2. Sold and censored examples under a 21-day censoring rule.

3.1 Slow21 gate definition

The Slow21 gate is a binary decision rule derived from time-to-sell: a listing is “slow” if it takes at least 21 days to sell. In hours, the script sets **SLOW21_H = 21 × 24 = 504**.

The AFT survival model produces a full conditional distribution for T. This enables computing Slow21 probability as a survival probability at 504 hours, rather than learning a separate classifier.

$$S(t \mid \mathbf{x}) = \mathbb{P}(T \geq t \mid \mathbf{x}) = 1 - F_D\left(\frac{\ln t - f(\mathbf{x})}{\sigma}\right)$$

Equation 2. Survival function under the AFT model; Slow21 probability is S(504h | x).

3.2 Censoring representation for XGBoost AFT

XGBoost’s AFT objective accepts censoring information via per-row bounds (label_lower_bound, label_upper_bound). An uncensored sold listing is encoded with lower=upper=duration_hours. A right-censored listing is encoded with lower=censor_time and upper=+∞ (implemented as a very large number).

$$p_i = \mathbb{P}(L_i \leq T_i \leq U_i \mid \mathbf{x}_i) = F_D\left(\frac{\ln U_i - f(\mathbf{x}_i)}{\sigma}\right) - F_D\left(\frac{\ln L_i - f(\mathbf{x}_i)}{\sigma}\right)$$

Equation 3. Likelihood contribution for interval-censored labels; right-censoring is the special case U=∞.

3.3 Train/eval splits

The script implements a temporally faithful evaluation by reserving a recent sold slice (e.g., the last *eval_days*=7 sold days) for evaluation. The calibration set (sval_days) can optionally be aligned to the same recent window. This mirrors operational deployment where the model must generalize to the most recent market regime.

4. Stock Features: Live Inventory as a Local Supply Signal

The stock features capture **competitive pressure** at the moment a listing is created/edited. If many similar listings are concurrently live in the same local market (super_metro) and device-generation segment, the time-to-sell distribution may shift upward (higher Slow21 probability).

4.1 Feature definitions

For each training row i with t0■, generation■, finn_id■, and super_metro_v4_geo■:

• **stock_n_sm4_gen**: number of listings live at t0■ within (super_metro_v4_geo, generation).

• **stock_n_sm4_gen_sbucket**: number of listings live at t0■ within (super_metro_v4_geo, generation, sbucket).

• **stock_share_sbucket**: ratio stock_n_sm4_gen_sbucket / NULLIF(stock_n_sm4_gen, 0).

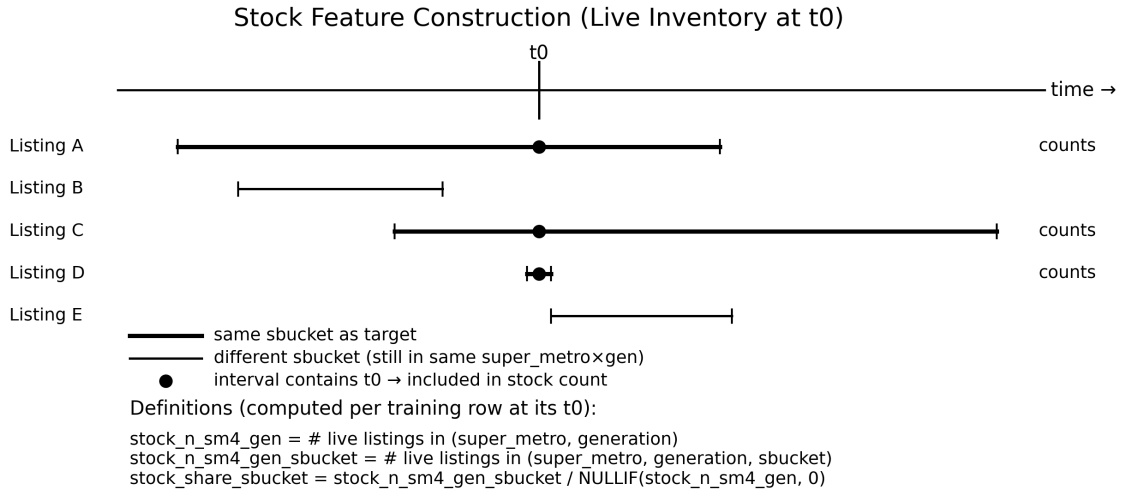

Figure 3. How the three stock features are computed from live intervals at t0.

4.2 Computing live intervals

A listing contributes to stock at t0 if it is observed to be live at that time. In historical data, "liveness" must be reconstructed. The script retrieves per-listing *first_seen* and an estimated *live_end*, where live_end is defined as the earlier of *last_seen* and *sold_date* when sold_date exists.

Important: last_seen and sold_date are used only to reconstruct whether a listing was live at a given historical timestamp. They are not exposed as direct model features. The final model receives only the three aggregate stock features above.

4.3 Algorithm: per-group sweep line counting

Naïvely counting live listings at t0 for every training row would be $O(N^2)$. Instead, the script computes counts per segmentation group using a sweep-line algorithm over interval endpoints. For each group g, we sort all interval starts and ends and compute the number of intervals containing each query time via prefix sums.

```
# Pseudocode (per group)
inputs:
  intervals: (start_ns, end_ns) for listings in group g
```

```
  queries: t0_ns for training rows in group g

events = []
for each interval:
  events.append((start_ns, +1))
  events.append((end_ns,   -1))   # treat end as inclusive in implementation

sort events by time
sort queries by time

active = 0
ei = 0
for each query time q in ascending order:
  while ei < len(events) and events[ei].time <= q:
      active += events[ei].delta
      ei += 1
  answer[q] = active
```

The implementation generalizes this approach to multiple grouping keys and uses stable merges to map group-level counts back to the original training rows. This yields approximately O((N+Q) log(N+Q)) per group, dominated by sorting.

5. Leakage Analysis of Stock Features

Because stock is constructed from longitudinal listing histories, it is necessary to be explicit about what constitutes leakage. Leakage occurs if any input feature depends on future information that would not be available at inference time. Formally, let ■(t0) denote the information set observable at t0. A feature ϕ is leak-free if ϕ = g(■(t0)) for some deterministic function g.

5.1 Why stock features can be leak-free

At inference time, one can (in principle) count how many similar listings are currently live on the marketplace site. Therefore the ideal stock feature is a function of the set of listings visible at t0. In training, we approximate this by reconstructing historical visibility from scrape logs.

The implementation uses last_seen and sold_date only to determine whether each other listing was live at the query time t0. This is a *measurement reconstruction step*, not a predictive feature. The model never receives last_seen or sold_date directly.

5.2 Practical risks and mitigations

• **Scrape cadence bias:** last_seen is a crawler artifact. If scrape frequency varies over time or by region, reconstructed liveness may be noisy. Mitigation: restrict to listings with both first_seen and last_seen, and consider minimum scrape coverage thresholds.

• **Self-counting:** the target listing may be included in the stock count (it is live at t0 by definition). This contributes an approximately constant +1 and is usually harmless, but can be removed by subtracting 1 when finn_id matches.

• **Outcome-dependent end times:** using sold_date to truncate live_end is appropriate for reconstructing visibility (a sold listing is not live after it sells). However, because sold_date is itself the event time, one must ensure the model is not inadvertently given any per-row signal derived from the target's sold_date.

• **Temporal confounding:** stock levels can covary with market-wide seasonality. The script's time-decay weighting and temporal evaluation window reduce—but do not eliminate—this risk.

In exploratory analysis, the correlation between stock and time (using 30-day buckets) was modest (≈0.14–0.20 depending on the stock variant), suggesting stock contains information beyond pure time trend. Nonetheless, the final verdict must come from strict temporal cross-validation or forward-chaining evaluation.

6. Statistical Model: Accelerated Failure Time (AFT)

The Accelerated Failure Time model assumes a log-linear relationship between covariates and the survival time. Instead of modeling the hazard directly (as in Cox models), AFT models the distribution of log time-to-event.

$$\ln T_i = f(\mathbf{x}_i) + \sigma \varepsilon_i, \quad \varepsilon_i \sim D$$

Equation 4. Core AFT model: a location-scale model for log time-to-sell.

Here, f(x) is a flexible function of covariates learned by gradient-boosted trees; σ>0 is a scale parameter (aft_scale); and the noise ε is drawn from a chosen distribution D (aft_dist ∈ {normal, logistic, extreme}).

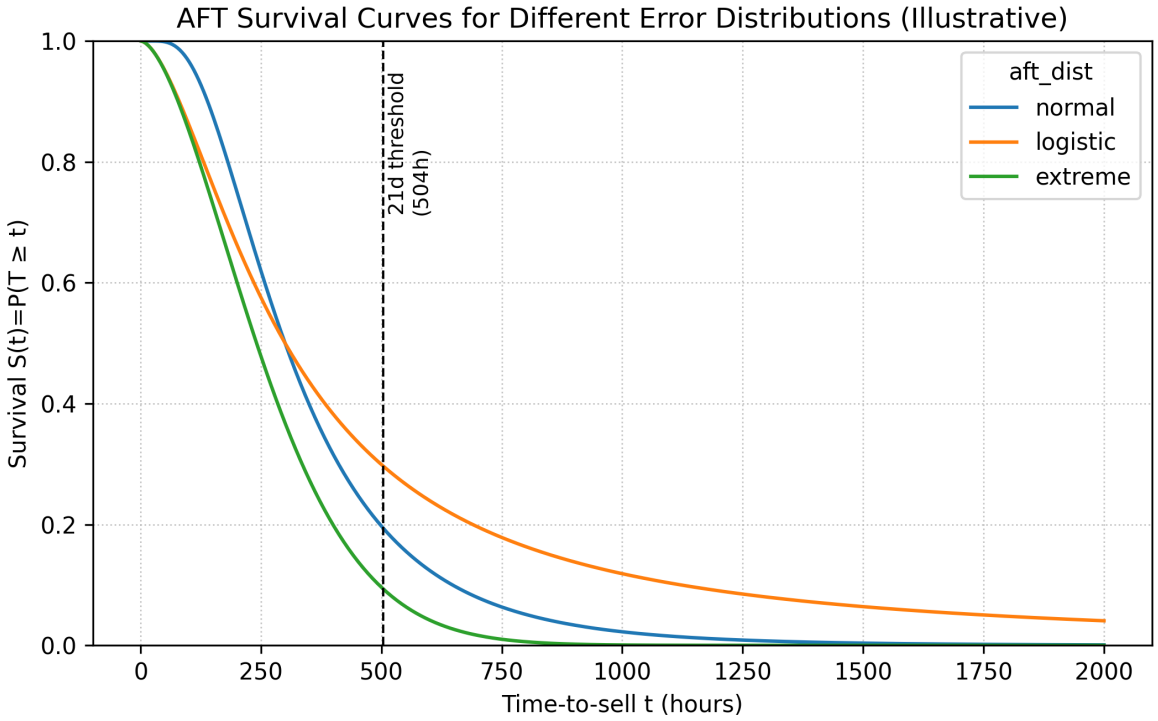

Figure 4. Illustrative survival curves for the supported AFT error distributions.

6.1 Handling censoring via likelihood bounds

For censored data, the exact event time is unknown. XGBoost represents each example i with an interval [L■, U■]. The AFT objective maximizes the probability mass of the event time falling in this interval under the model. Uncensored sold examples have L■=U■=duration_hours; right-censored examples have U■=∞.

6.2 From survival model to Slow21 probabilities

Because the model provides a full conditional survival function S(t|x), we can compute Slow21 probability exactly as S(504h|x). Operationally, the script also evaluates predictions by converting predicted time-to-sell into a hard decision at 504 hours. This produces metrics such as precision/recall/F1 on the Slow21 classification task.

7. Algorithmic Model: XGBoost AFT with Leaf-wise GBDT

The function f(x) in the AFT model is learned as a gradient-boosted ensemble of decision trees. Each tree is a piecewise-constant function that partitions the feature space and assigns an additive correction to the model's prediction.

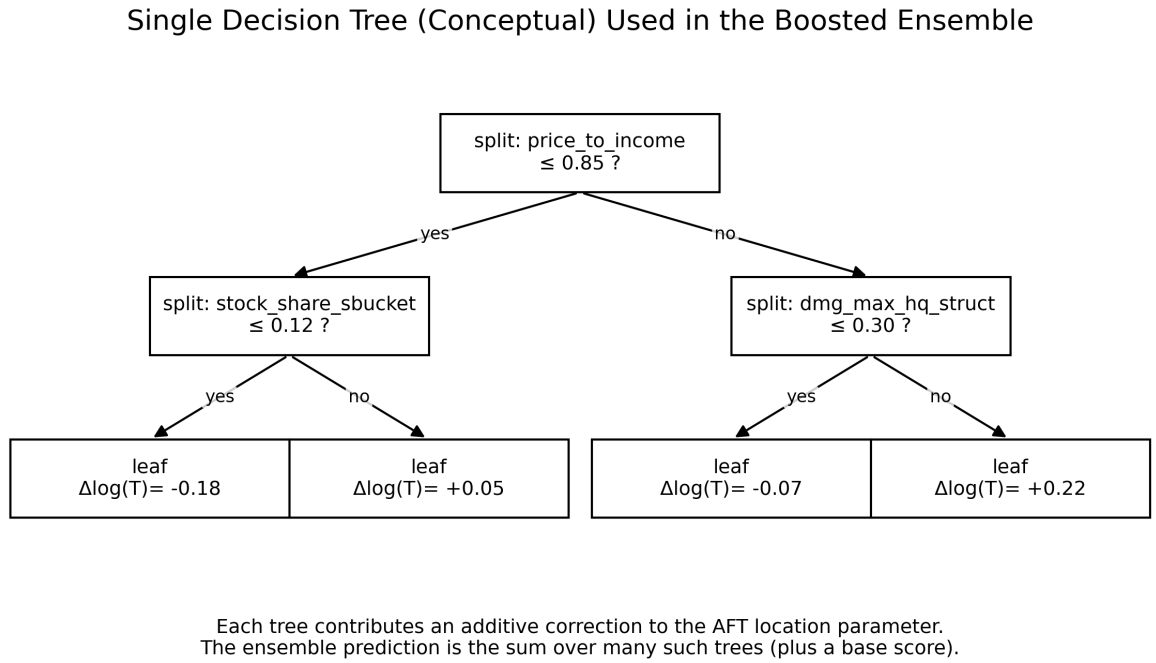

Figure 5. A single decision tree; in practice the model uses hundreds to thousands of trees.

7.1 Boosting as stage-wise functional optimization

Boosting constructs a sequence of models F■, F■, …, F_T where each new tree is trained to reduce the current loss. For differentiable losses, a tree is fit to the negative gradient (residual) of the loss with respect to the current predictions.

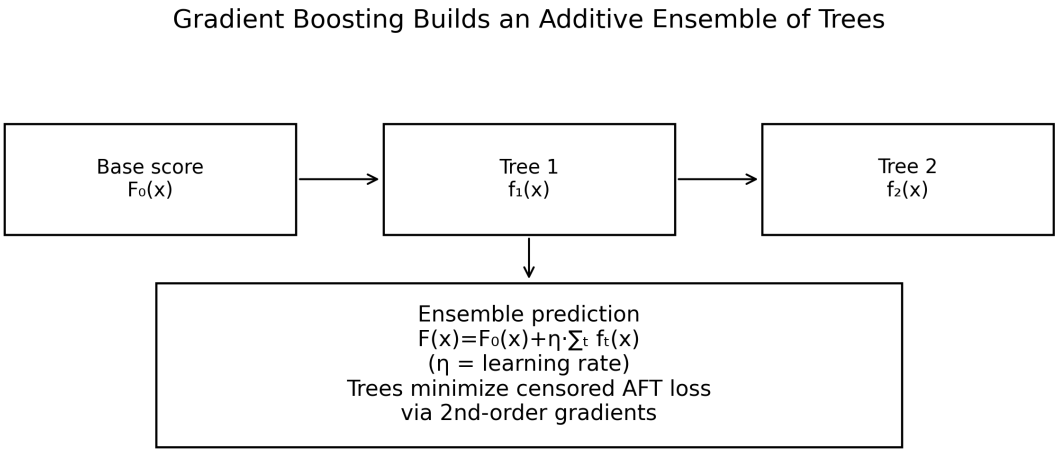

Figure 6. The additive nature of gradient boosting (η is the learning rate).

7.2 How XGBoost selects splits

XGBoost uses a second-order Taylor approximation of the objective and greedily chooses splits that maximize the reduction in the approximate loss. For a candidate split that partitions a node into left/right children with gradient sums G_L, G_R and Hessian sums H_L, H_R, the split gain takes the following canonical form:

$$\text{Gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}\right) - \gamma$$

Equation 5. Canonical split gain formula in second-order gradient boosting.

The optimal leaf weight for a region is similarly obtained by minimizing the second-order approximation, giving:

$$w^\star = -\frac{G}{H+\lambda}$$

Equation 6. Optimal leaf weight under L2 regularization.

In the AFT setting, gradients and Hessians come from the censored negative log-likelihood rather than squared error, but the tree-building mechanics remain the same: compute per-row derivatives, aggregate by candidate split, and select the split with highest gain.

7.3 Leaf-wise growth (grow_policy=lossguide)

The script configures XGBoost with grow_policy=lossguide and max_leaves=num_leaves. This produces a leaf-wise growth pattern analogous to LightGBM: the algorithm repeatedly splits the leaf that yields the largest gain until the leaf budget is exhausted. This can represent complex interactions with fewer trees, but it

also demands careful regularization.

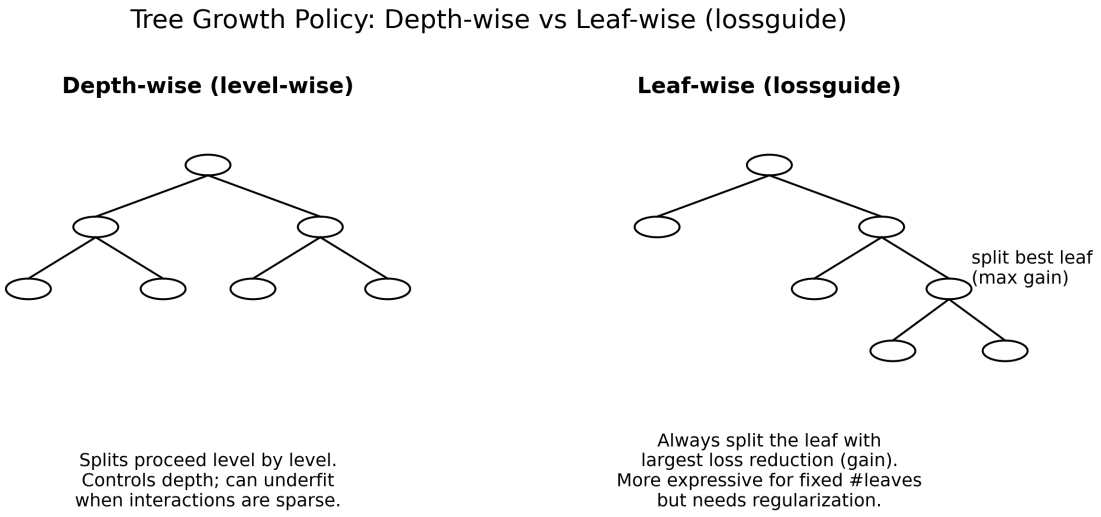

Figure 7. Depth-wise vs. leaf-wise growth. The training script uses leaf-wise growth (lossguide).

8. Objective Shaping: Weights, Tails, and Boundary Focus

A pure AFT likelihood objective optimizes the overall log-likelihood of survival times. However, the operational decision boundary is the Slow21 threshold (504h), and the error profile around this boundary matters disproportionately. The training script therefore applies a principled **sample-weight shaping** strategy that retains a survival objective while emphasizing the regimes that most affect Slow21 quality.

8.1 Time-decay weighting (recency)

To adapt to market drift, the script down-weights older observations using an exponential half-life schedule:

$$w_i = 0.5^{\text{age}_i/\text{half\_life}}$$

Equation 7. Time-decay weighting used throughout training.

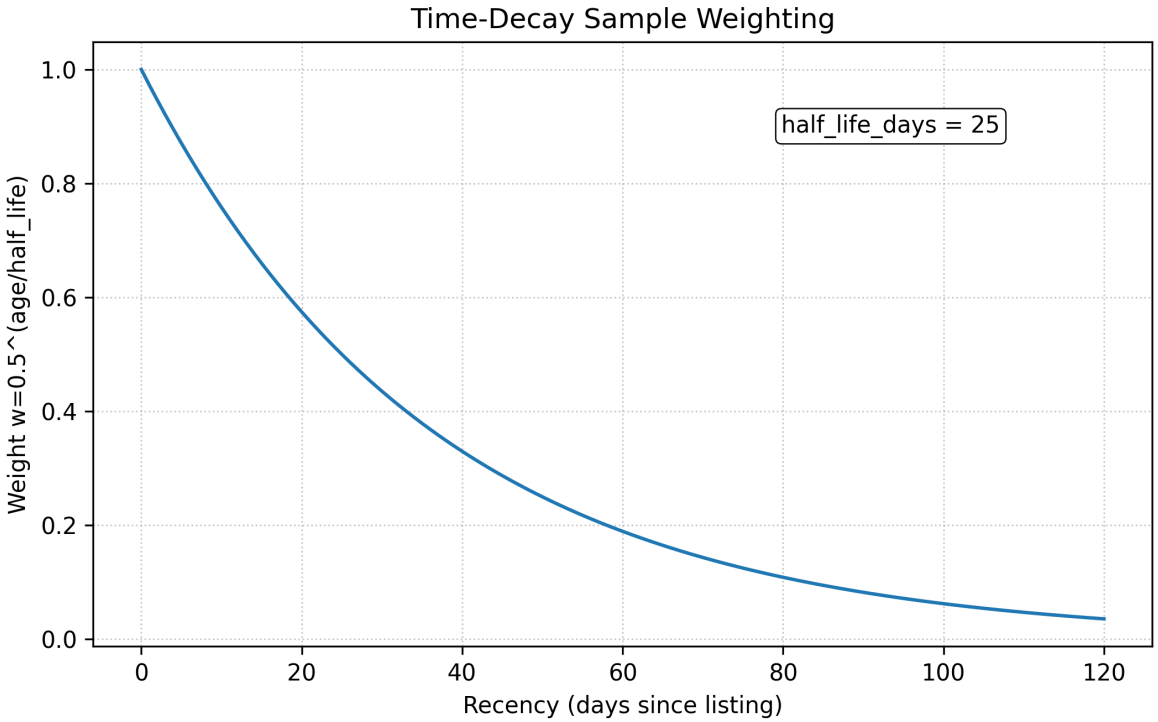

Figure 8. Example time-decay curve for half_life_days=25.

8.2 Tail upweighting for slow listings

The script multiplies sample weights for sold listings with durations above 7 days (168h) and above 21 days (504h). This increases gradient signal in the slow tail where classification errors are most costly.

8.3 Boundary-focus weighting

To further concentrate learning near the 21-day decision boundary, the script applies a smooth, monotone boundary multiplier based on a sigmoid centered at 504h with width controlled by boundary_focus_sigma (in days).

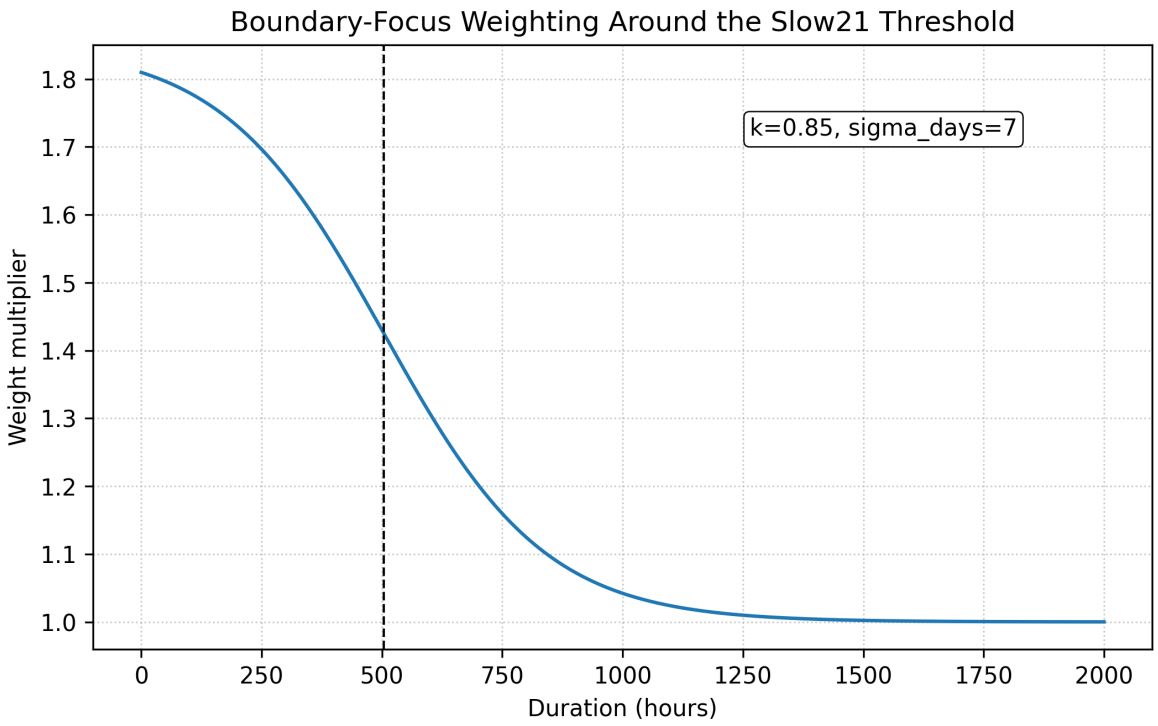

Figure 9. Boundary-focus multiplier (k=0.85, sigma_days=7).

8.4 Guardrail on very fast sellers

A mild additional multiplier is applied to very fast sellers (<10 days) to reduce instabilities in certain derived metrics (e.g., SAC_LT10). This is a pragmatic stabilizer and should be revisited if the evaluation regime changes.

9. Post-Training Calibration

Even with a well-specified survival objective, tree ensembles may produce predictions that are systematically biased in certain regions of the feature space. The script therefore performs a final **monotone calibration** step using isotonic regression on a recent sold slice.

Calibration is applied after training (and after hyperparameter selection) to map raw model predictions to empirically calibrated predictions. The design is explicitly boundary-aware: it prioritizes fidelity around the Slow21 boundary while preserving monotonicity.

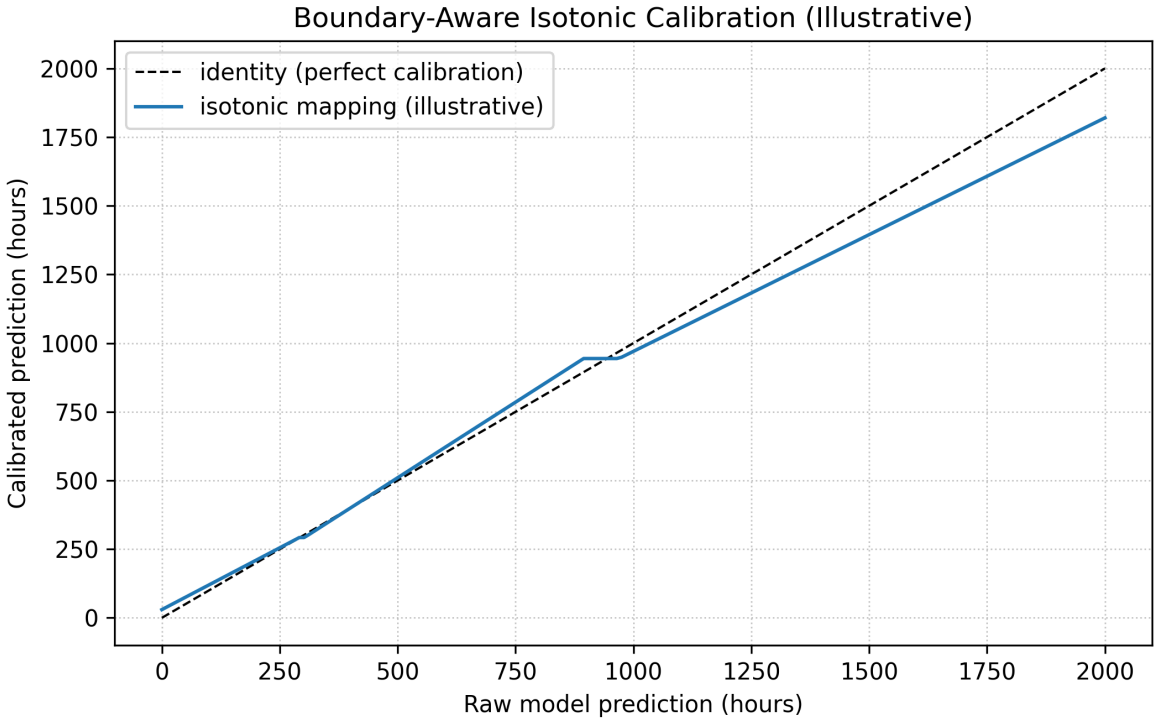

Figure 10. Illustrative isotonic mapping from raw to calibrated predictions.

In deployment terms, calibration reduces the risk that a fixed threshold (e.g., 504h) yields drifting precision/recall as market conditions evolve. Because isotonic regression is non-parametric and monotone, it is robust to misspecification but should be trained on sufficiently recent and representative data.

10. WOE Anchor: A Lightweight, Interpretable Baseline Signal

In addition to the main AFT model, the script computes a **Weight-of-Evidence (WOE) anchor** for Slow21. This anchor is a compact, interpretable score derived from a small set of anchor-related variables (e.g., price deltas vs sold anchors, presentation metrics). It is then injected back as two columns: *woe_anchor_p_slow21* and *woe_anchor_logit_slow21*.

10.1 Why include WOE?

WOE transforms discrete bands of a variable into log-odds contributions. It offers three practical advantages:

• **Interpretability:** bands have explicit log-odds meaning and can be audited directly in SQL.

• **Robustness:** a low-capacity baseline can stabilize the full model when feature stores are missing or shift.

• **Deployment simplicity:** WOE tables can be persisted and applied consistently in batch or online scoring.

10.2 Leakage-safe OOF scoring

To prevent information bleed from the evaluation horizon, the script supports out-of-fold (OOF) scoring for the WOE anchor using time-based folds (by t0 day). The WOE mapping is learned on K−1 folds and applied to the held-out fold, producing OOF predictions for training rows.

Artifacts—including band cuts, WOE maps, and (optionally) OOF scores—can be persisted to Postgres to support reproducibility and downstream auditing.

11. Explainability: SHAP and Feature-Block Attribution

Tree ensembles are powerful but can be opaque. The pipeline therefore computes SHAP (SHapley Additive exPlanations) feature importance and persists results to Postgres. SHAP provides a principled decomposition of each prediction into per-feature contributions that sum to the model output.

Beyond individual features, the implementation tracks a **feature_block** metadata tag and aggregates SHAP mean(|value|) by block. This block-level view is particularly valuable when the feature space spans multiple stores and derived pipelines.

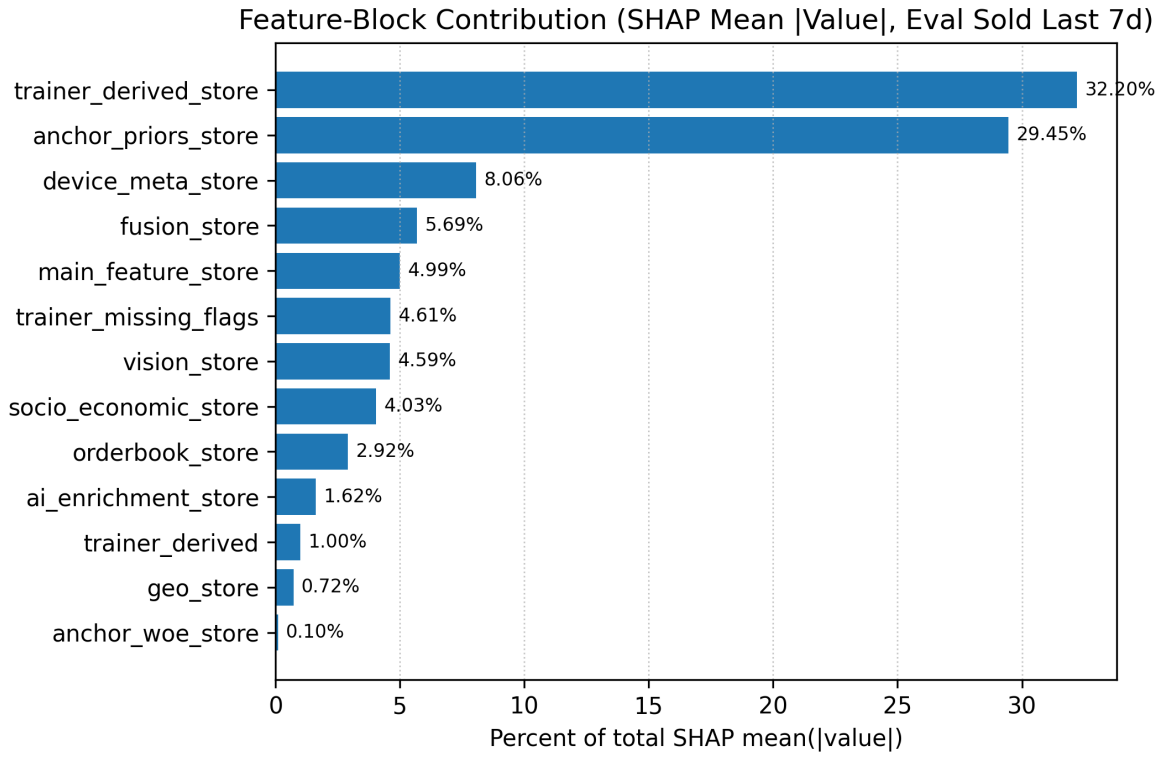

Figure 11. Example feature-block attribution from an evaluation slice (percent of total SHAP mean(|value|)).

In the shown example, trainer-derived and anchor-prior features dominate the explanation mass, while vision/fusion/device-meta provide smaller but non-negligible contributions. This is consistent with the hypothesis that time-to-sell is strongly driven by price anchoring and local market conditions, with visual condition signals refining the tail risk.

12. Empirical Evidence: Stock Features Carry Conditional Signal

Before investing in full retraining runs, it is useful to validate that stock features have standalone predictive signal and that the signal is not purely a proxy for time trends. The following results summarize exploratory SQL experiments on a sample of ~18.3k sold listings with stock computed at t0 and conditioned on price and market flow quintiles.

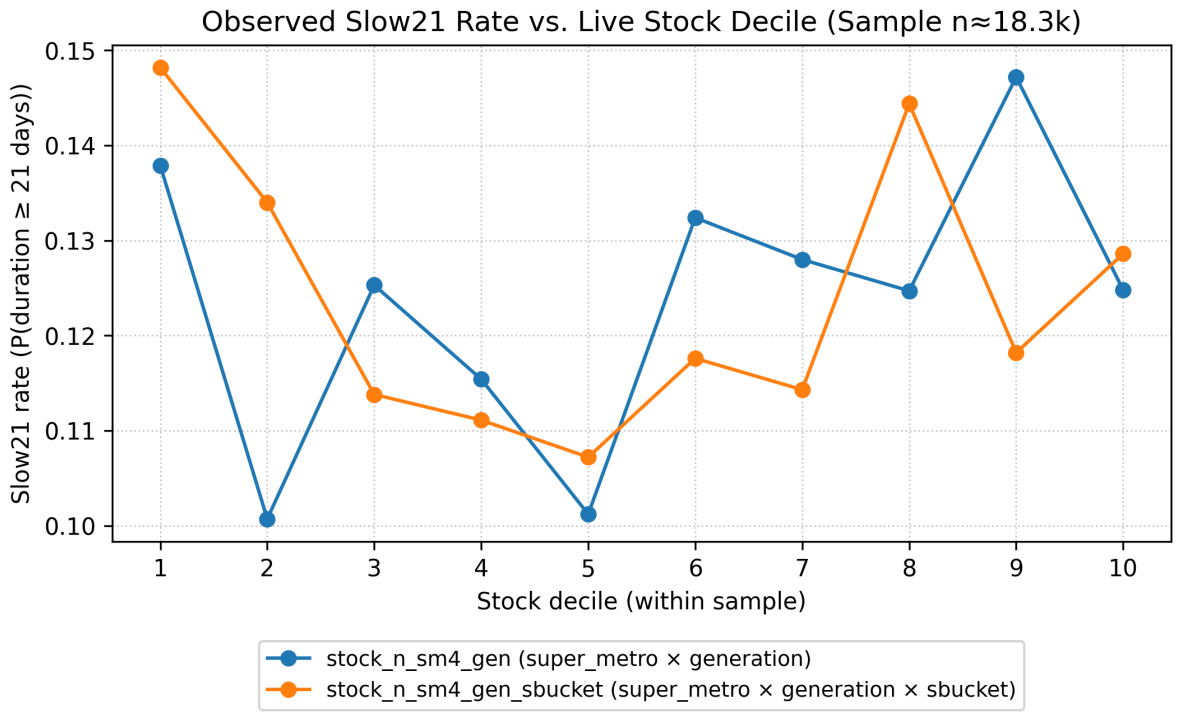

Figure 12. Observed Slow21 rate by decile of stock features (univariate view).

The univariate relationship is non-monotone (as is common in competitive markets), suggesting interactions with price positioning, flow, and other segment attributes. Therefore, conditional analysis is more informative than global correlation.

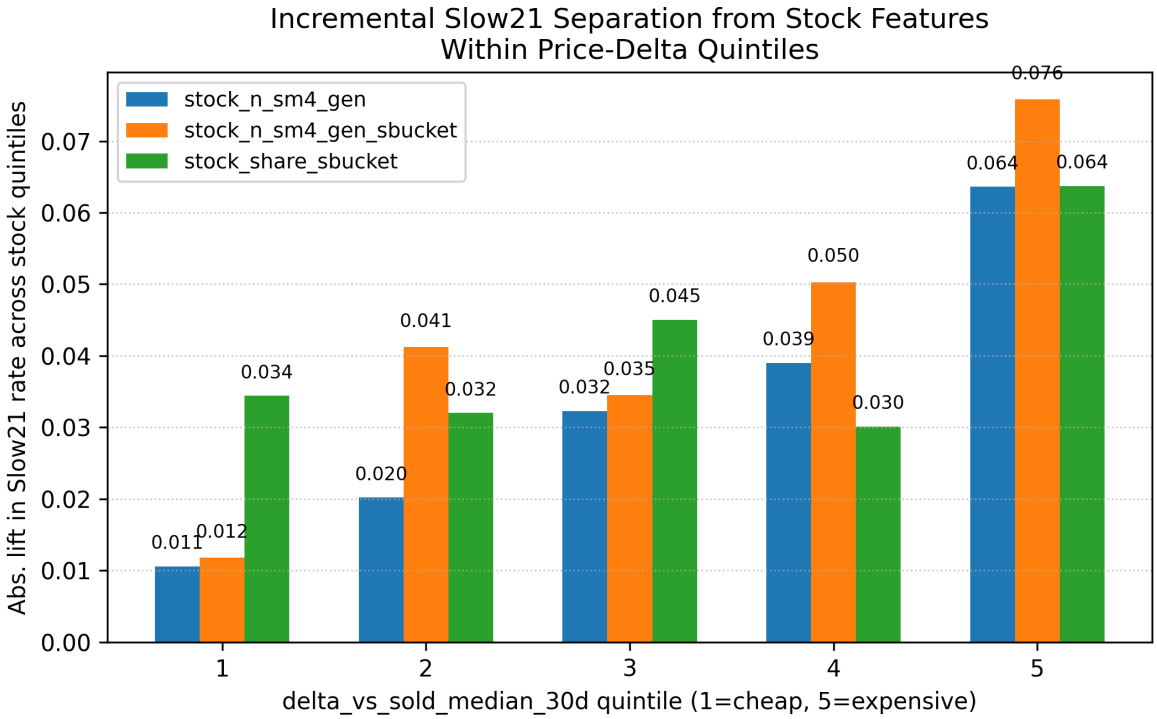

Figure 13. Conditional separation in Slow21 rate across stock quintiles within price-delta quintiles.

The largest incremental separation occurs at higher price-delta quintiles (more expensive vs the sold median), where stock pressure plausibly increases the chance of a slow sale.

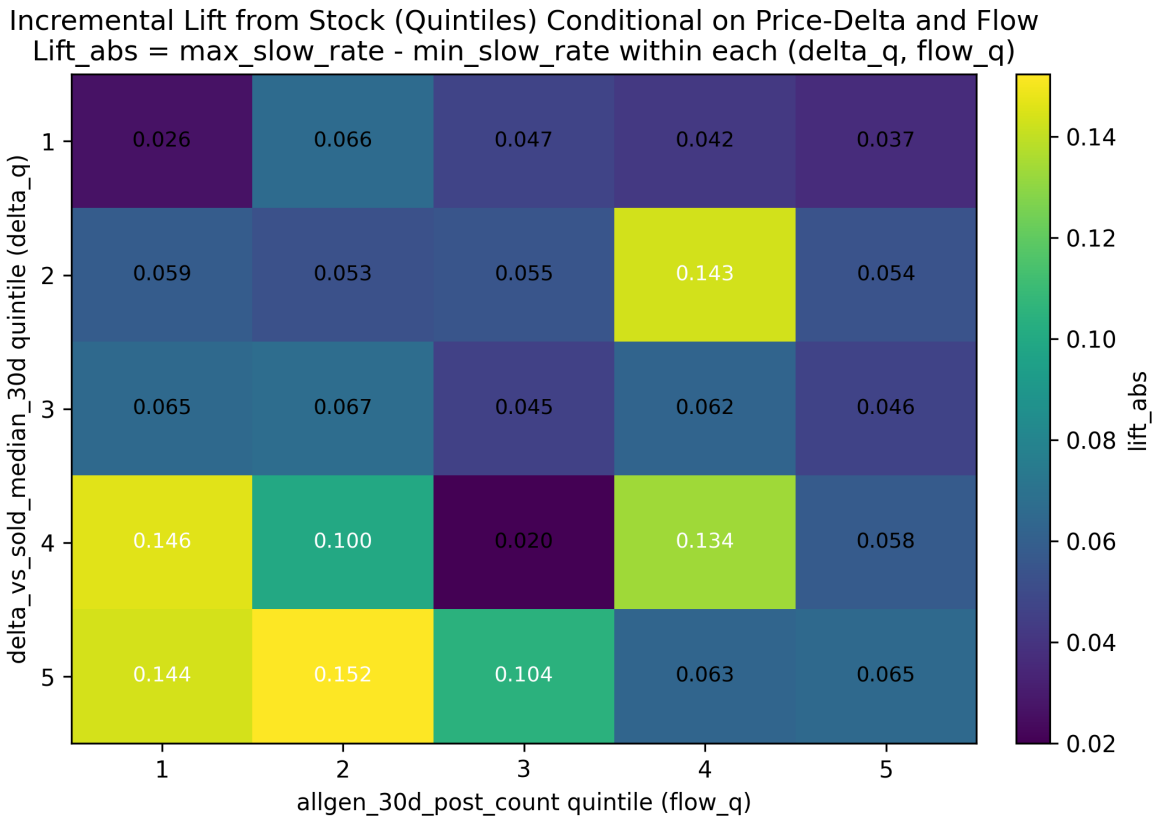

Figure 14. Conditional lift heatmap: stock is most informative in specific (delta_q, flow_q) regions.

Several segments show substantial lift_abs (>0.10), indicating that stock meaningfully reshapes the Slow21 probability mass in those segments.

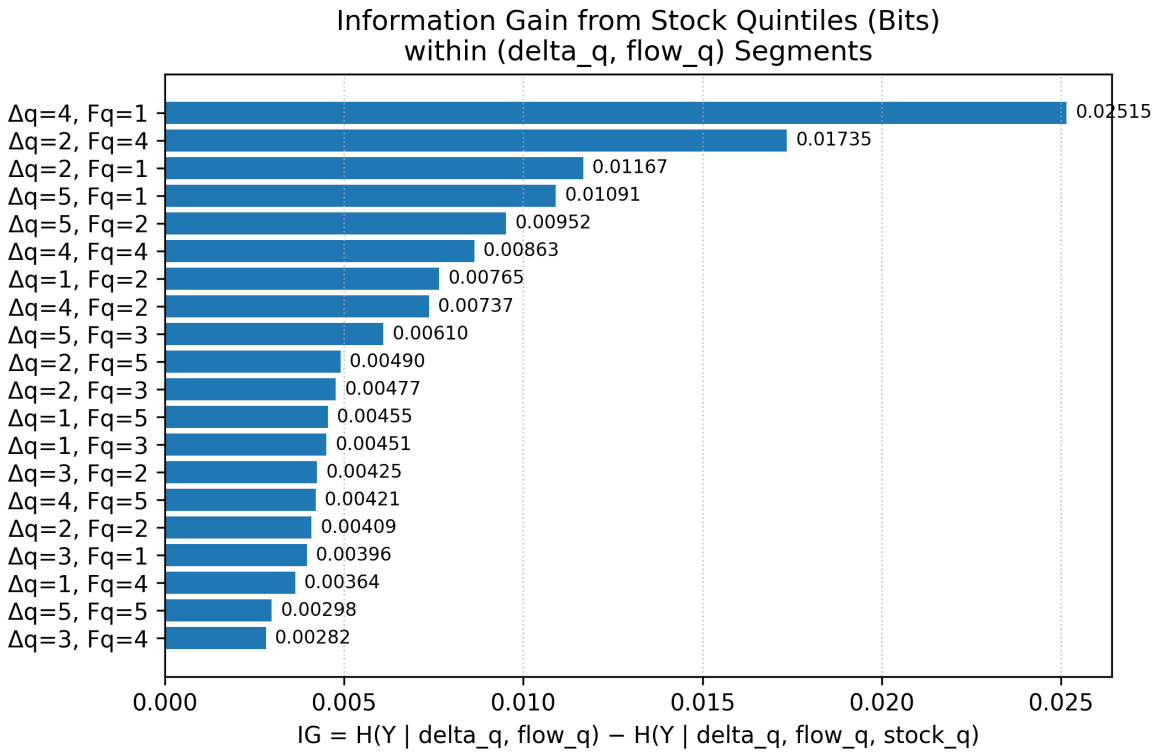

Figure 15. Information gain from stock quintiles, by (delta_q, flow_q) segment.

Information gain (in bits) provides a scale-free measure of how much stock reduces uncertainty about the Slow21 outcome within a segment. The highest-IG segments are typically those where both price positioning and local flow create multiple plausible outcomes, so stock acts as a tie-breaker.

13. Implementation Notes and Reproducibility

13.1 Enabling stock features via CLI

The training script exposes a boolean CLI flag **--use_stock_features**. When enabled, the pipeline computes stock features in Python and adds them to the model feature matrix. In logs, you should see a line similar to:

```
[train_aft] [stock] merged (+3 cols): stock_n_sm4_gen_sbucket, stock_n_sm4_gen, stock_share
```

If this line is absent, stock features are not being added (either the flag was not passed, or the data required for computation was unavailable).

13.2 Parameter mapping (LightGBM-style knobs → XGBoost)

The script presents a LightGBM-like hyperparameter interface but trains with XGBoost. The most important mappings are:

CLI knob	XGBoost param	Meaning
--n_estimators	num_boost_round	Number of boosting iterations (trees)
--learning_rate	learning_rate (η)	Shrinkage applied to each new tree
--num_leaves	max_leaves (lossguide)	Leaf budget per tree (leaf-wise growth)
--min_data_in_leaf	min_child_weight (proxy)	Regularizes small leaves / minimum effective sample mass
--feature_fraction	colsample_bytree	Column subsampling per tree
--bagging_fraction	subsample	Row subsampling per tree
--lambda_l2	reg_lambda	L2 regularization on leaf weights
--aft_scale	aft_loss_distribution_scale	Scale parameter σ in AFT
--aft_dist	aft_loss_distribution	Error distribution: normal/logistic/extreme

13.3 Pandas SQL warning

The script uses pandas.read_sql_query with a psycopg connection object. Pandas emits a warning recommending SQLAlchemy; this warning is benign and does not affect correctness. Converting to SQLAlchemy is an optional cleanup for maintainability.

14. Recommended Validation and Ablation Plan

Exploratory signal checks are necessary but not sufficient. To validate that stock features genuinely improve generalization, we recommend the following experiment design:

• **Strict temporal split:** train on older sold + censored data; evaluate on the last 7–14 sold days (no mixing).

• **Two-model ablation:** train (A) baseline without stock and (B) with stock, holding all other settings fixed; compare Slow21 F1 and calibration error.

• **Stability checks:** evaluate by super_metro, generation, and sbucket to ensure improvements are not isolated to a single segment.

• **Leakage probe:** run a "future-shift" test where t0 is artificially advanced; if metrics jump abnormally, stock reconstruction likely leaks.

• **Permutation importance:** permute stock within (time_bucket, super_metro) to measure incremental value beyond time/geo proxies.

Only after these checks should stock features be promoted from Python-only to a certified DB feature store (MV/view), so that training and production use identical computation and the signal can be monitored operationally.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). *Optuna: A Next-generation Hyperparameter Optimization Framework.*

Chen, T. & Guestrin, C. (2016). *XGBoost: A Scalable Tree Boosting System*. KDD.

Ke, G. et al. (2017). *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*. NeurIPS.

Lundberg, S. M. & Lee, S.-I. (2017). *A Unified Approach to Interpreting Model Predictions*. NeurIPS.

Klein, J. P. & Moeschberger, M. L. (2003). *Survival Analysis: Techniques for Censored and Truncated Data*. Springer.

Zadrozny, B. & Elkan, C. (2002). *Transforming Classifier Scores into Accurate Multiclass Probability Estimates* (isotonic calibration).