Hypothesis-Driven Time Series Forecasting

Abstract

Kaggle competitions are often won by practitioners who squeeze the last fraction of a percent out of exotic architectures. This whitepaper takes a different approach: it's a **process write-up**, not a leaderboard postmortem. Using the Hedge Fund Time Series Forecasting competition as a case study, we document the exploration loop—hypothesis, experiment, takeaway—that led to key insights about **low signal-to-noise (SNR) prediction problems**.

Core Finding: In environments where "predict zero" is a strong baseline, **robust objectives and careful calibration matter more than model complexity**. The iterative discipline of small, one-change-at-a-time ablations consistently outperformed kitchen-sink feature engineering.

1. Competition Context

1.1 The Problem

The competition tasked participants with predicting a continuous, return-like target for multiple entities across four forecast horizons (1, 3, 10, and 25 time steps ahead). This is a classic multi-horizon time series problem, but with a twist: **extremely low signal-to-noise ratio**.

Key Characteristics:

- **86 anonymized features** with no domain semantics
- **Multiple entity hierarchies** (`code`, `sub_code`, `sub_category`)
- **Four distinct horizons** with different predictability profiles
- **Weighted samples** contributing unevenly to the final score

1.2 The Metric Trap

The competition used a weighted, clipped RMSE-style "skill score." This metric has critical implications:

1. **Variance is penalized heavily** — predictions with high variance can score worse than predicting the mean
2. **Clipping limits upside** — heroic predictions on outliers don't help
3. **Weights matter** — high-weight samples dominate the score

The Baseline Revelation: Early experiments showed that **predicting zero for all rows** was a surprisingly strong baseline. This meant any model needed to demonstrate clear value-add over doing nothing—a humbling starting point.

1.3 The Leakage Constraint

Predictions for time index `t` could only use data available up to `t`. This sequential constraint ruled out many standard preprocessing techniques (global normalization, future-aware feature engineering) and forced a disciplined approach to validation.

---

2. Experimental Philosophy

Before diving into results, it's worth stating the philosophy that guided this work:

2.1 One Change at a Time

Every experiment modified exactly one variable from the previous state. This discipline creates a clear causal chain: if the metric improves, we know why. Kitchen-sink approaches that change five things at once make it impossible to attribute gains.

2.2 Hypothesis-First Experimentation

Each script began with a written hypothesis. Not "let's try XGBoost" but "I believe the MSE loss is unstable due to outliers; a Huber objective should reduce variance." This forces clarity of thought and creates a documentary trail.

2.3 Respecting Small Deltas

The public leaderboard only reflects part of the test set. We treated "small CV gains" (< 0.005) as fragile and potentially spurious. Robust improvements had to be reproducible across multiple time-series splits.

---

3. Experiment Timeline

Phase A: EDA and Baseline Scaffolding

Goal: Understand the data shape, horizon differences, and establish reliable training/validation infrastructure.

Key Activities:

- `01_data_exploration.py` : Target distribution analysis, feature correlation heatmaps, horizon-specific behavior profiling
- `02_lgb_baseline.py` through `04_lgb_baseline_v3.py` : Iterating on LightGBM pipeline plumbing

Observations: The target distribution was approximately symmetric around zero with heavy tails. Different horizons showed markedly different predictability:

Horizon	Target Std	Naive R^2	Notes
H1	0.032	0.003	Extremely noisy
H3	0.041	0.008	Slightly more signal
H10	0.058	0.021	Best predictability
H25	0.072	0.015	High variance, moderate signal

Takeaway: The baseline modeling problem is dominated by outliers and noise. Naive loss choices (MSE) led to unstable training.

Phase B: Weighting Experiments

Hypothesis: Because the competition metric is weighted, training should respect the `weight` column (without using it as a feature).

Experiment (`05_high_weight_focus.py`): Tested "focus on high-weight rows" approaches—upsampling high-weight instances, using `weight` directly as sample weights in LightGBM.

Results:

Strategy	CV Score	Notes
Uniform weights	0.0523	Baseline
Raw weights	0.0498	Slight improvement
High-weight only (top 20%)	0.0612	Hurt generalization
`sqrt(weight + 1)`	0.0487	Best compromise

Takeaway: Over-focusing on high-weight samples collapsed generalization. A monotone transform (`sqrt(weight + 1)`) reduced extreme weight dominance while still respecting the metric. The key insight: **moderation in weighting prevents overfitting to a few samples**.

Phase C: Robust Objectives (Outlier Resistance)

Hypothesis: Outliers cause MSE-trained models to "swing too hard"—large predictions that hurt the clipped skill score. A robust loss should reduce predictions variance.

Experiments (`06_strategy_refinement_v2.py`, `08_advanced_tuning.py`): Explored Huber loss variants and tuned their aggressiveness (the `alpha` parameter, which controls the "box width" for quadratic vs. linear behavior).

Alpha Sweep Results:

Huber Alpha	CV Score	Prediction Std
0.5	0.0521	0.018
1.0	0.0492	0.014
1.5	0.0478	0.012
2.0	0.0483	0.011
3.0	0.0501	0.009

Takeaway: There's a sweet spot. Too tight (low alpha) and the model loses sensitivity; too loose and you're back to MSE instability. The optimal alpha was horizon-dependent, with shorter horizons preferring tighter boxes.

Key Insight: Robust losses reduce variance and produce predictions that are easier to calibrate under the skill-score metric.

Phase D: Calibration via Shrinkage (Variance Control)

Hypothesis: Even robust models remain overconfident. Shrinking predictions toward zero can improve the metric by reducing variance further—essentially admitting "I don't know" when confidence is low.

Implementation: Global and per-horizon shrinkage factors, tuned alongside the model:

```
def apply_shrinkage(preds, horizon, shrinkage_config):
    factor = shrinkage_config[horizon]
    return preds * factor

# Example shrinkage config
SHRINKAGE = {
    'H1': 0.12,   # Shrink most aggressively (lowest signal)
    'H3': 0.06,   # Very conservative
    'H10': 0.27,  # More confident
```

```
    'H25': 0.29   # Longest horizon, moderate confidence
}
```

The Configuration Fragility Lesson:

This phase taught us the most painful lesson of the competition. Shrinkage is **extremely sensitive**, especially by horizon. A single wrong factor can dominate results:

Horizon	Correct Factors	Incorrect Factors	Score Impact
H1	0.12	0.15	-0.003
H3	0.06	0.15 (too high)	**-0.021**
H10	0.27	0.28	-0.001
H25	0.29	0.30	-0.002

A single misconfiguration in H3 shrinkage wiped out weeks of gains from other optimizations.

Takeaway: Treat shrinkage like a first-class hyperparameter. Log it per run, per horizon. Automate validation against a holdout set before submission.

Phase E: Feature Set Ablations (Macro vs. Micro)

Hypothesis: Aggregated "market/sector" features might help longer horizons by adding macro context. The anonymized features represent micro (entity-level) signals; perhaps sector-level means would add regime information.

Experiments (`09_feature_engineering.py` , `10_hybrid_submission.py`):

- Created rolling aggregates by `code` and `sub_category`
- Tested lag features at 1, 3, 5, 10 periods
- Compared "macro-enriched" vs. "raw features only"

Results:

Feature Set	CV Score	Test Score
Raw (86 features)	0.0478	0.0512
+ Rolling aggregates	0.0461	0.0534
+ Lag features	0.0455	0.0548
+ Both	0.0449	0.0567

The Gap: Note the divergence between CV and test. **Feature engineering improved cross-validation while hurting out-of-sample performance.**

Root Cause Analysis: The test period likely represented a different market regime. Features that captured "how things usually behave" became liabilities when the market shifted. The rolling aggregates, in particular, encoded historical correlations that didn't persist.

Takeaway: In low-SNR, regime-shifting domains, **simpler and more robust feature sets outperform sophisticated engineering**. The test set penalized cleverness.

Phase F: Ensembling and Strategy Benchmarks

Goal: Test whether modest complexity (blends) can add robustness without overfitting.

Experiments (`11_ensemble_strategy.py`, `15_advanced_score_push.py`):

Benchmarked multiple strategies against the best single-model baseline:

Strategy	CV Score	Notes
LightGBM (tuned)	0.0478	Single-model baseline
CatBoost (tuned)	0.0485	Slightly worse
LGB + CatBoost (0.7/0.3)	0.0471	Best blend
XGBoost addition	0.0476	No improvement

Strategy	CV Score	Notes
Ridge fallback	0.0498	Stable but weak
10-seed ensemble	0.0474	Reduced variance slightly
Quantile regression	0.0512	Didn't help

What Did Work:

- LightGBM + CatBoost blend at 70/30 was the most promising, particularly on H25 (long horizon)
- Multi-seed ensembling reduced variance slightly but didn't improve mean performance

What Didn't Work:

- Adding a third model (XGBoost) added noise without benefit
- Quantile regression experiments failed to improve calibration
- Residual boosting (stacking errors) overfit badly

Takeaway: The bar for "complexity that helps" is extremely high in low-SNR settings. Improvements, when found, tend to be **horizon-specific** and easy to overfit without careful validation.

---

4. Key Takeaways

4.1 In Low-SNR Settings, Calibration Beats Capacity

Model complexity (deeper trees, more features, larger ensembles) doesn't help when the signal is weak. Instead, **robust objectives + careful calibration** dominated our improvements. The ability to shrink predictions intelligently—admitting uncertainty—was worth more than any feature engineering.

4.2 Weighting Needs Moderation

Sample weights in the training objective should reflect the competition metric, but not naively. Transforming weights (`sqrt` , `log`) prevented the model from overfitting to a handful of high-weight outliers.

4.3 Horizon Behavior Differs

"One size fits all" configurations hide failures. Each horizon had different optimal:

- Shrinkage factors
- Huber alpha values
- Feature importance profiles

Treating horizons independently—or at least monitoring them separately—was essential.

4.4 CV is Necessary but Not Sufficient

Time-series cross-validation with strict leakage controls was mandatory. But small CV deltas (< 0.005) were often noise. We learned to distrust "improvements" that didn't survive multiple validation splits.

4.5 Feature Engineering Can Hurt

In regime-shifting environments, features that encode historical patterns become liabilities. Simpler feature sets were more robust to distribution shift between train and test.

---

5. Process Artifacts: Using This Repo as a Template

The accompanying repository is structured as a **process artifact**, not just a code dump.

5.1 Entry Points for Learning

Script	What It Demonstrates
`01_data_exploration.py`	Initial EDA, target analysis, sanity checks
`06_strategy_refinement_v2.py`	The "robust loss + calibration" pivot

Script	What It Demonstrates
`08_advanced_tuning.py`	Huber alpha and shrinkage tuning
`09_feature_engineering.py`	Ablation methodology (and the regime-shift lesson)
`15_advanced_score_push.py`	Structured strategy benchmarking

5.2 The REPORT.md Convention

Each phase is documented in `REPORT.md` with:

- **Hypothesis:** What we believed going in
- **Experiment:** What we actually tried
- **Takeaway:** What we learned (including failures)

This format forces intellectual honesty. You can't hide behind "I tried a lot of things."

5.3 GEMINI.md as Change Log

Lightweight project notes track every significant decision. When you're three weeks in and can't remember why you chose `alpha=1.5`, the answer is documented.

---

6. Conclusion

Kaggle competitions reward practitioners who understand their problem deeply. This case study demonstrates that in **low signal-to-noise forecasting**:

1. **Robust losses** (Huber) reduce prediction variance
2. **Calibration** (shrinkage) admits uncertainty intelligently
3. **Simple features** are more robust to regime shifts
4. **One-change-at-a-time ablations** build causal understanding
5. **Horizon-specific tuning** captures divergent dynamics

The most important lesson isn't any specific technique—it's the **discipline of hypothesis-driven experimentation**. Write down what you believe, test it, and document what you learned. Over time, this creates a flywheel of compounding knowledge that makes each competition easier than the last.

Appendix A: Shrinkage Configuration Reference

```
# Final shrinkage configuration
# Arrived at through grid search on holdout set

SHRINKAGE_CONFIG = {
    'H1': 0.12,   # 88% shrinkage toward zero
    'H3': 0.06,   # 94% shrinkage (lowest signal horizon)
    'H10': 0.27,  # 73% shrinkage
    'H25': 0.29   # 71% shrinkage
}

# Application
def calibrate_predictions(df, shrinkage_config):
    """Apply horizon-specific shrinkage to predictions."""
    df = df.copy()
    for horizon in ['H1', 'H3', 'H10', 'H25']:
        mask = df['horizon'] == int(horizon[1:])
        df.loc[mask, 'prediction'] *= shrinkage_config[horizon]
    return df
```

Appendix B: Validation Split Strategy

```
# Time-series cross-validation
# Respects temporal ordering, no shuffling

def create_time_splits(df, n_splits=5):
    """
    Create time-aware train/val splits.
    Each split uses earlier data for training,
    later data for validation.
    """
```

```
    unique_times = sorted(df['ts_index'].unique())
    split_points = np.linspace(
        len(unique_times) * 0.5,  # Start at 50% mark
        len(unique_times) * 0.9,  # End at 90% mark
        n_splits
    ).astype(int)

    splits = []
    for i, split_idx in enumerate(split_points):
        cutoff_time = unique_times[split_idx]
        train_mask = df['ts_index'] < cutoff_time
        val_mask = (df['ts_index'] >= cutoff_time) & \
                   (df['ts_index'] < unique_times[min(split_idx + 20,
len(unique_times)-1)])
        splits.append((train_mask, val_mask))

    return splits
```

Resources

- [Competition Repository]
- [LightGBM Documentation]
- [Huber Loss Explained]