

Beating a 0.964 Baseline: What I Learned Building a Stacking Ensemble for Smartphone Addiction Prediction

Kaggle Playground Series S6E8 | Gradient boosting, stacking, pseudo-labeling, and the experiments that failed

When I started this competition, I assumed the path to a better score was simple: add more features, add more models, stack everything. I was wrong about half of that. This post covers what worked, what didn't, and what I would do differently.

The Problem

The task was to predict whether a person is addicted to their smartphone (**addicted_label**) from behavioral data: daily screen time, social media hours, gaming hours, sleep, notifications, app opens, stress level, and a few demographic columns. The metric was **ROC-AUC**.

The scale made it interesting: **691,369 training rows** and **296,302 test rows**. At this size, small gains matter, and every experiment costs real GPU time.

Step 1: Look at the Data Before Modeling

Before any modeling, I checked shapes, data types, duplicates, target balance, and missing values. The key finding was that **every feature had missing values, between 4% and 19%**.

Instead of just imputing them away, I treated missingness as a signal. I added a missing-flag per column and a total-missing count. Tree models like LightGBM, XGBoost, and CatBoost also handle NaNs natively, so I left the raw values untouched.

Step 2: Behavioral Feature Engineering

Raw hour counts don't capture behavior patterns well, so I built ratio and interaction features:

- **Social and gaming share** of total screen time
- **Screen-to-sleep ratio**
- **Weekend vs. weekday drift** in screen time
- **Notifications per app open**
- **Leisure vs. work** hours

This gave me a 35-feature baseline.

Step 3: A Clean Baseline with Leakage-Safe Validation

I trained LightGBM, XGBoost, and CatBoost on GPU with 5-fold Stratified K-Fold and early stopping. The most important habit here was using **out-of-fold (OOF) predictions**. OOF AUC is a leakage-free estimate of how the model will behave on unseen data, so I could trust my comparisons.

Baseline result: ~0.9639 OOF AUC.

Step 4: Experiments That Did Not Help

This is the part most blog posts skip.

More features didn't win. I added KMeans behavioral clusters, distance-to-centroid, and fold-safe target encoding of categoricals (35 to 40 features). The OOF AUC did not beat the baseline, so I went back to the original set. More features are not automatically better.

Stacking similar models barely helped. A logistic-regression meta-model over the three tree models scored 0.96432, only a small gain over the 0.96389 baseline. The reason was clear in the correlation matrix: the three tree models were **98-99.6% correlated**. They were finding the same patterns, so there was nothing for a stack to combine.

A neural network added diversity. I trained a residual MLP in PyTorch, and its predictions correlated much less with the trees (0.93-0.95). That is what a stack needs, models that make different mistakes. (A practical note: I ran the NN on CPU after hitting GPU compatibility issues on the platform.)

Step 5: What Actually Moved the Score

Two changes gave the largest gains, both inspired by the Kaggle community discussion for this competition:

- 1 **A residual feature:** $\text{other_screen_time} = \text{daily_screen_time} - (\text{social} + \text{gaming} + \text{work/study})$. It captures screen time the individual components can't explain.
- 2 **Value-level target encoding:** encoding the exact rounded value of numeric columns (not just a category) using smoothed target statistics. I computed it strictly inside each fold, so the validation fold never sees its own labels.

Combined with higher model capacity (255 leaves, a lower learning rate of 0.02, up to 4000 rounds with early stopping), this pushed the models to:

Model	OOF AUC
LightGBM	0.96766
XGBoost	0.96773
CatBoost	0.96687
Final stack (LGB + XGB + CAT)	0.96777

Step 6: Multi-Seed Bagging and Pseudo-Labeling

For the final push, I did two things:

- **Pseudo-labeling:** I took test rows where the model was very confident (prediction above 0.97 or below 0.03). That was 47.9% of the test set, about 142,000 rows. I added them to the training data as extra labeled examples. Importantly, OOF AUC was still measured **only on real training rows**, so the score stays honest.
- **Multi-seed bagging:** I trained LightGBM with three different seeds and averaged them to reduce variance.

Final bagged LightGBM OOF AUC: 0.96799.

What I Learned

- 1 **Validate before you celebrate.** OOF scoring let me reject ideas (clusters, extra encodings) that felt smart but didn't help.
- 2 **Diversity beats quantity in ensembles.** Three nearly identical models are one model. A different architecture helped more than another tree model would have.
- 3 **Missing values can be information.** Treating missingness as a feature is cheap and often useful.
- 4 **Leakage is easy to introduce.** Target encoding and pseudo-labeling both need careful, fold-safe handling.
- 5 **Compute is a real constraint.** Some folds took 4-5 minutes each. Planning experiments around GPU time mattered as much as the modeling ideas.
- 6 **Mistakes happen, and fixing them is part of the work.** One run stopped after 4 of 5 folds, and I had to correct the test-prediction scaling and recompute the AUC on the completed folds. Catching that matters more than hiding it.

What I'd Try Next

- Better hyperparameter tuning (Optuna) instead of manual settings
- Feature selection to cut redundant engineered features
- A properly tuned NN included in the final blend
- Calibration checks on the pseudo-labeled predictions

Resources

- **Kaggle notebook:** [Click Here](#)
- **Competition:** Kaggle Playground Series S6E8

If you're working on a similar tabular problem, I'd love to hear what worked for you. Feel free to reach out via the contact page.