**10th Place Solution - ~350 oofs => 9 hillclimbing versions => Final Autogluon ensemble**

**Introduction**

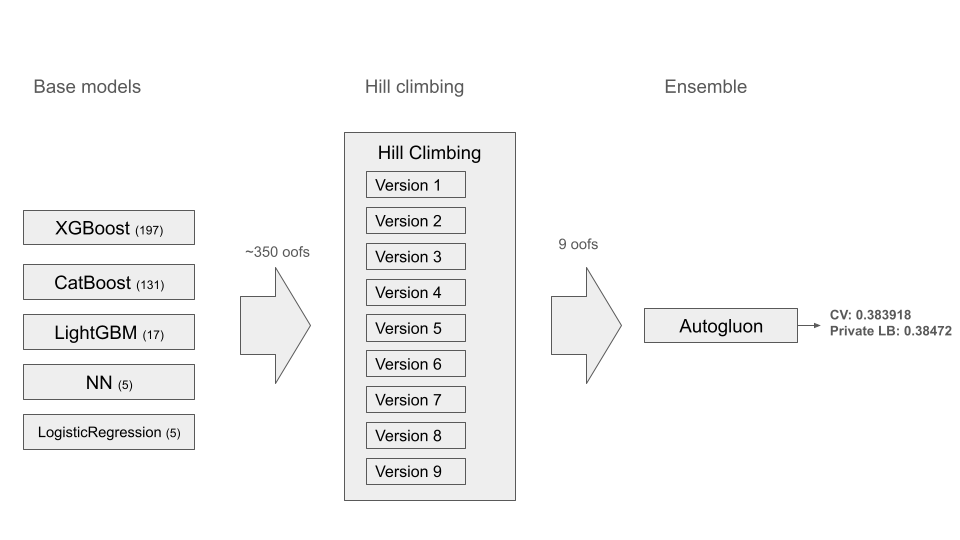
First and foremost, I want to extend a massive thank you to Kaggle for hosting the competition and to the community for sharing knowledge and creating interesting discussions. This was my first time working with the map@3 metric, and I found that ranking predictions give room for more creativity in the solutions.

Congratulations to the maestro himself [@cdeotte](https://www.kaggle.com/cdeotte) for winning yet another playground competition and to [@masayakawamata](https://www.kaggle.com/masayakawamata), [@mahoganybuttstrings](https://www.kaggle.com/mahoganybuttstrings), [@hahahaj](https://www.kaggle.com/hahahaj) and [@optimistix](https://www.kaggle.com/optimistix) for giving him a run for his money.

**TL;DR**

* Used standard models: XGBoost, CatBoost, LightGBM, NN, and Logistic Regression with minimal feature engineering.
* Generated ~350 Out-of-Fold (OOF) predictions. The best single model was an XGBoost with CV=0.37799, tuned with Optuna, using 6 fixed folds and sample weight bumping.
* Created 20 versions of hill climbing ensembles, with the best achieving a CV=0.383830.
* Used AutoGluon to ensemble the 9 best hill climbing results, reaching a final CV of 0.383918 and a Private LB score of 0.38472.

**Solution Overview**

  
Each new version of hillclimbing had a larger set of oofs and the last one had ~350.

**Base Models**

**XGBoost:**

XGBoost performed the best, with the top single model achieving a CV=0.37799. Thanks to [@ravi20076](https://www.kaggle.com/ravi20076) for pointing out [here](https://www.kaggle.com/competitions/playground-series-s5e6/discussion/582644) in the early stage of the competition. The best parameters found were:

Params = {

'max\_depth': 17,

'min\_child\_weight' : 4,

'subsample' : 0.8932774942420912,

'colsample\_bytree' : 0.40035993059700825,

'gamma' : 0.44089845615519774,

'reg\_alpha' : 3.0189326628791378,

'reg\_lambda' : 1.0463133005441632,

'max\_delta\_step': 5}

This was achieved using sample weight bumping for 2nd, 3rd, and 4th places as described below. It is surprising that a depth of 17 was optimal, which I attribute to the synthetic nature of the data.

**LightGBM/CatBoost:**

Both were included for diversity, but they generally performed worse than XGBoost.

**Neural Networks (NN):**

I used a modified version of the excellent notebook by [@paddykb](https://www.kaggle.com/paddykb): [No Keras, No Loan](https://www.kaggle.com/code/paddykb/ps-s4e10-no-keras-no-loan-cv-0-963). The best I could achieve was CV=0.35097.  
I tried both cases using numeric features as numeric and categorical, and found the latter to perform best.

**Logistic Regression:**

This was adapted from [@siukeitin](https://www.kaggle.com/siukeitin)'s [What if you can only use logistic regression…](https://www.kaggle.com/competitions/playground-series-s5e6/discussion/585144). With sample weight bumping, I achieved a CV=0.37468.

**Ensembling Strategy**

**Hill Climbing:**

I created 20 different versions using the GPU hill-climbing method by [@cdeotte](https://www.kaggle.com/cdeotte). My best hill climbing ensemble consisted of the following 14 OOF predictions:

- oof\_model-1-15-w11-xgboost\_trial\_3\_map3\_0.37799.npy

- oof\_model-13-3-w10-logisticregression\_trial\_0\_map3\_0.37530.npy

- oof\_model-4-3-w10-NN\_trial\_0\_map3\_0.35016.npy

- oof\_model-5-5-w10-catboost\_trial\_1\_map3\_0.34875.npy

- oof\_model-12-1-w10-xgboost\_index\_0\_map3\_0.36555.npy

- oof\_model-1-15-w11-xgboost\_trial\_0\_map3\_0.37737.npy

- oof\_model-4-2-w10-NN\_trial\_3\_map3\_0.34908.npy

- oof\_model-5-5-w10-catboost\_trial\_2\_map3\_0.35196.npy

- oof\_model-12-1-w10-xgboost\_index\_67\_map3\_0.36579.npy

- oof\_model-1-7-w10-xgboost\_trial\_0\_map3\_0.37640.npy

- oof\_model-13-1-w10-logisticsregression\_trial\_0\_map3\_0.37411.npy

- oof\_model-12-1-w10-xgboost\_index\_39\_map3\_0.36198.npy

- oof\_model-3-2-w10-lightgbm\_trial\_2\_map3\_0.35041.npy

- oof\_model-5-3-w10-catboost\_trial\_2\_map3\_0.34606.npy

This resulted in CV=0.383830 and Public LB=0.38268. As [@cdeotte](https://www.kaggle.com/cdeotte) has pointed out several times, the key is diversity in the models rather than high-performing single models.

*On a side note, I think there is more to investigate with hill climbing (ideas, not used in this competition):*

* Hill climb each fold separately, apply to the test set, and average the results.
* Experiment with starting the hill climb from different models, not just the best-performing one.
* Implement a tree-based search (e.g., using Alpha-beta pruning) to find an optimal set of models, rather than relying on a purely greedy approach.

During this competition, I noticed that sometimes adding a new OOF to the hill climb would significantly worsen the score, which makes me curious about potential improvements to the method.

**Final Ensemble with AutoGluon:**

The 9 best hill climbing ensembles were further ensembled using AutoGluon to produce the final submission with CV=0.383918 and Private LB=0.38472.  
Thanks to [@ravaghi](https://www.kaggle.com/ravaghi) for pointing out how to use map@3 in AutoGluon [here](https://www.kaggle.com/competitions/playground-series-s5e6/discussion/582846).

**Other Techniques**

**1. Use Sample\_Weight x4 for original data**

Following a common technique from public notebooks, I set the sample\_weight of the original data to 4x, which improved the score.

**2. Use Sample\_Weight to bump up 2nd, 3rd, and 4th placements**

I analyzed OOF predictions to identify instances where the correct class was predicted as 2nd, 3rd, or 4th. I then retrained the models with increased sample\_weight for these specific instances to encourage the model to rank them higher. The potential gain for moving up one place is:

* 2nd => 1st : 0.5 (from 0.5 to 1.0)
* 3rd => 2nd : 0.17 (from 0.33 to 0.5)
* 4th => 3rd : 0.33 (from 0.0 to 0.33)

I experimented with bumping the sample\_weight for predictions where the probability difference between the two places where below a threshold (e.g., 0.02). This technique yielded a ~0.001 CV improvement for many models. I tried bumping only the 2nd place predictions, as well as bumping all three ranks. I considered automating this with Optuna, but that will be a project for a future playground competition.

**3. Using the same CV folds for everything**

I used a fixed 6 Stratified-K-Fold strategy for all experiments. To ensure consistency and prevent data leakage between stages, I added a verification step to my scripts:

def verify\_first\_fold(val\_idx):

# Hardcoded first 10 indices of the first validation fold

val\_idx\_expected = np.array([1, 2, 5, 8, 9, 14, 16, 23, 27, 30])

if np.array\_equal(val\_idx[:10], val\_idx\_expected):

return

print("First fold validation indices do not match the expected values.")

print("Expected:", val\_idx\_expected)

print("Actual:", val\_idx[:10])

raise ValueError("Fold mismatch detected. Halting execution.")

# In the training loop

for fold, (train\_idx, val\_idx) in enumerate(kf.split(X=train\_df, y=...)):

if fold == 0:

verify\_first\_fold(val\_idx)

This is not a bulletproof method, but it saved me from polluting my OOFs on several occasions.

**4. Optuna - Tricks**

* I consolidated all hyperparameter tuning runs in the Optuna Dashboard for better tracking. All the results are stored in local sqlite3 databases for each notebook and I use a separate consolidation script to merge into a single db:

def merge\_optuna\_studies(

source\_storages: List[str],

output\_storage: str,

min\_trials: int = 10,

name\_prefix: str = ""

) -> int:

"""Merge studies from multiple Optuna storages into one."""

copied\_studies = 0

for source\_storage in source\_storages:

try:

# Get all study summaries from the source storage

study\_summaries = optuna.study.get\_all\_study\_summaries(storage=source\_storage)

if len(study\_summaries) == 0:

print(f"No studies found in {source\_storage}. Skipping.")

continue

print(f"Found {len(study\_summaries)} studies in {source\_storage}")

# Process each study

for summary in study\_summaries:

study\_name = summary.study\_name

n\_trials = summary.n\_trials

if n\_trials < min\_trials:

print(f"Skipping study '{study\_name}' with only {n\_trials} trials (minimum required: {min\_trials})")

continue

print(f"Copying study '{study\_name}' with {n\_trials} trials")

optuna.study.copy\_study(from\_study\_name=study\_name,

from\_storage=source\_storage,

to\_storage=output\_storage,

to\_study\_name=study\_name)

copied\_studies += 1

except Exception as e:

print(f"Error processing {source\_storage}: {str(e)}")

return copied\_studies

* I used WilcoxonPruner to reduce computational effort, though this resulted in fewer OOFs being generated. Here is the setup:

# Pruner setup

p\_threshold = 0.08 # p-value threshold for pruning

n\_startup\_steps = 2 # Number of trials to complete before pruning begins

pruner = optuna.pruners.WilcoxonPruner(

p\_threshold=p\_threshold,

n\_startup\_steps=n\_startup\_steps

)

# Create the Optuna study

study = optuna.create\_study(

direction="maximize",

pruner=pruner

)

def objective(trial):

# ...

# Within the CV loop, report the intermediate score to Optuna

# ... train and evaluate on fold ...

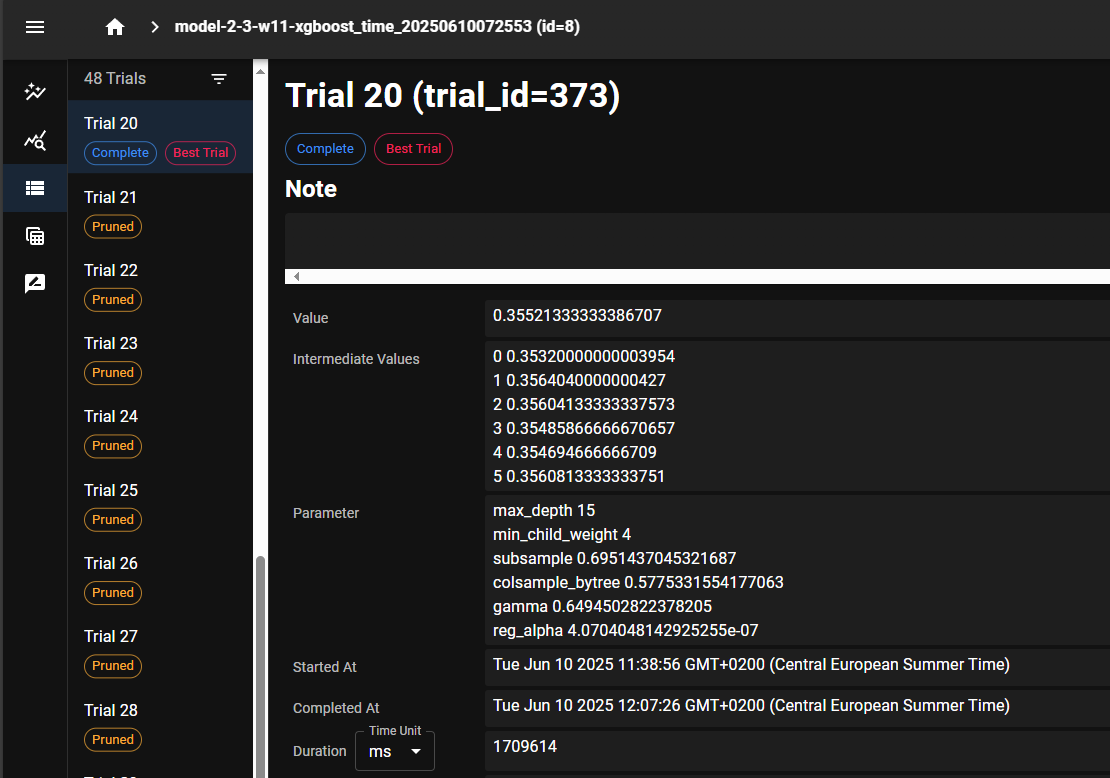
trial.report(map3\_score, step=fold)

# Prune the trial if it is unpromising unless on the last fold.

if trial.should\_prune() and fold < total\_folds - 1:

raise optuna.TrialPruned()

return final\_cv\_score

This setup also populates the CV scores (intermediate Values) values in the Optuna Dashboard:  


* I always enqueued default parameters or other strong baseline configurations as the first trial.

**5. Feature Engineering Search**

Mid-competition, I fixed my best XGBoost parameters and systematically tested ~200 engineered features, calculating the map@3 for each. Some examples:

{

"tested\_features": {

"Potassium\_binned": 0.36687777777820074,

"Humidity\_x\_Moisture\_x\_Potassium": 0.36101355555603537,

"Humidity\_x\_Potassium": 0.36245444444491326,

"Humidity\_x\_Moisture": 0.36123777777825505,

"log\_Temparature": 0.3660744444448719,

"Temparature\_x\_Nitrogen\_div\_Humidity": 0.36036377777826734,

"sqrt\_Phosphorous": 0.36733622222264173,

"Temparature\_x\_Humidity\_div\_Nitrogen": 0.3603313333338274,

**Not a single feature improved the result over the baseline without feature engineering.** This is consistent with other competitors' findings. However, some of these OOFs were still selected by the hill climbing algorithm due to the diversity they added. I am aware of [@cdeotte](https://www.kaggle.com/cdeotte)'s advice to add all features and then remove them one by one, but I lacked the computational resources for that approach. I had to remove some of the oofs due to memory limitation of the number of oofs possible to hillclimb on my pc.

**What Did Not Work / Future Improvements**

**TabTransformer:**

I spent some time trying to get a network based on the [TabTransformer](https://github.com/lucidrains/tab-transformer-pytorch" \t "_blank) library to work, but could only achieve a CV=0.31. I later discovered the excellent notebook by [@omidbaghchehsaraei](https://www.kaggle.com/omidbaghchehsaraei) [TabTransformer](https://www.kaggle.com/code/omidbaghchehsaraei/tabtransformer-cv-0-35327-lb-0-36542" \t "_blank) but did not have time to incorporate his work.

**Identify DAP/Urea in a separate model:**

Analyzing the per-class map@3 score from several of my OOFs, I found that DAP and Urea had poor results compared to other classes:

CLASS-WISE MAP@3 CONTRIBUTION: (final oof)  
Class 14-25-14: MAP@3=0.432626, Contribution=0.066011 (114,436 samples)  
Class 10-26-26: MAP@3=0.408642, Contribution=0.062052 (113,887 samples)  
Class 17-17-17: MAP@3=0.426334, Contribution=0.063923 (112,453 samples)  
Class 28-28: MAP@3=0.372506, Contribution=0.055209 (111,158 samples)  
Class 20-20: MAP@3=0.369162, Contribution=0.054581 (110,889 samples)  
Class DAP: MAP@3=0.341869, Contribution=0.043240 (94,860 samples)  
Class Urea: MAP@3=0.315330, Contribution=0.038814 (92,317 samples)

I tried to train a separate binary classification model to identify DAP/Urea and use its predictions as a feature for the main models, but this did not improve my score. I also tried using sample\_weight to specifically bump up DAP/Urea, but that only resulted in a much worse CV.

**Lessons Learned**

* Trust your local CV.
* Prioritize model diversity for ensembling.
* Learn from discussion and other notebooks
* I used the same seed for all models. However, as [@cdeotte](https://www.kaggle.com/cdeotte) mentioned in his [1st place solution](https://www.kaggle.com/competitions/playground-series-s5e6/discussion/587393), map@3 can be sensitive to randomness in training, so perhaps there was an element of luck involved. :)

This was my first attempt at a write-up. Thank you to the Kaggle Community for all the lessons learned, and I'm looking forward to the next playground competition. Happy Kaggling!