Kaggler's Guide to LightGBM Hyperparameter Tuning with Optuna in 2021

Squeeze every bit of performance out of your LightGBM model

Bex T. Sep 3 · 7 min read ⋆

Comprehensive tutorial on LightGBM hyperparameters and how to tune them using Optuna.

Photo by Pixabay on Pexels. All images are by the author unless specified otherwise.

Introduction

In the previous article, we talked about the basics of LightGBM and creating LGBM models that beat XGBoost in almost every aspect. This article focuses

on the last stage of any machine learning project — hyperparameter tuning (if we omit model ensembling).

First, we will look at the most important LGBM hyperparameters, grouped by their impact level and area. Then, we will see a hands-on example of tuning LGBM parameters using Optuna — the next-generation bayesian hyperparameter tuning framework.

Most importantly, we will do this in a similar way to how top Kagglers tune their LGBM models that achieve impressive results.

> *I highly suggest reading the first part of the article if you are new to LGBM. Although I will briefly explain how Optuna works, I also recommend reading my separate post on it to get the best out this article.*

You Are Missing Out on LightGBM. It Crushes XGBoost in Every Aspect
Edit description
towardsdatascience.com

Why Is Everyone at Kaggle Obsessed with Optuna For Hyperparameter Tuning?
Edit description
towardsdatascience.com

> *Get the notebook of the article here on Kaggle.*

Overview of the most important parameters

Generally, hyperparameters of the most tree-based models can be grouped into 4 categories:

1. Parameters that affect the structure and learning of the decision trees
2. Parameters that affect the training speed
3. Parameters for better accuracy
4. Parameters to combat overfitting

Most of the time, these categories have a lot of overlap, and increasing efficiency in one may risk a decrease in another. That's why tuning them manually is a giant mistake and should be avoided.

Frameworks like Optuna can automatically find the "sweet medium" between these categories if given a good enough parameter grid.

Hyperparameters that control the tree structure

> *If you are not familiar with decision trees, check out this legendary video by StatQuest.*

In LGBM, the most important parameter to control the tree structure is `num_leaves`. As the name suggests, it controls the number of decision leaves in a single tree. The decision leaf of a tree is the node where the 'actual decision' happens.

The next is `max_depth`. The higher `max_depth`, the more levels the tree has, which makes it more complex and prone to overfit. Too low, and you will underfit. Even though it sounds hard, it is the easiest parameter to tune — just choose a value between 3 and 12 (this range tends to work well on Kaggle for any dataset).

Tuning `num_leaves` can also be easy once you determine `max_depth`. There is a simple formula given in LGBM documentation - the maximum limit to `num_leaves` should be `2^(max_depth)`. This means the optimal value for `num_leaves` lies within the range (2^3, 2^12) or (8, 4096).

However, `num_leaves` impacts the learning in LGBM more than `max_depth`. This means you need to specify a more conservative search range like (20, 3000) - that's what I mostly do.

Another important structural parameter for a tree is `min_data_in_leaf`. Its magnitude is also correlated to whether you overfit or not. In simple terms, `min_data_in_leaf` specifies the minimum number of observations that fit the decision criteria in a leaf.

For example, if the decision leaf checks whether one feature is greater than, let's say, 13 — setting `min_data_in_leaf` to 100 means we want to evaluate this leaf only if at least 100 training observations are bigger than 13. This is the gist in my lay terms.

The optimal value for `min_data_in_leaf` depends on the number of training samples and `num_leaves`. For large datasets, set a value in hundreds or thousands.

Check out this section of the LGBM documentation for more details.

Hyperparameters for better accuracy

A common strategy for achieving higher accuracy is to use many decision trees and decrease the learning rate. In other words, find the best combination of `n_estimators` and `learning_rate` in LGBM.

`n_estimators` controls the number of decision trees while `learning_rate` is the step size parameter of the gradient descent.

Ensembles like LGBM build trees in iterations, and each new tree is used to correct the "errors" of the previous trees. This approach is fast and powerful, and prone to overfitting.

That's why gradient boosted ensembles have a `learning_rate` parameter that controls the learning speed. Typical values lie within 0.01 and 0.3, but it is possible to go beyond these, especially towards 0.

So, the perfect setup for these 2 parameters (`n_estimators` and `learning_rate`) is to use many trees with early stopping and set a low value for `learning_rate`. We will see an example later.

You can also increase `max_bin` than the default (255) but again, at the risk of overfitting.

Check out this section of the LGBM documentation for more details.

More hyperparameters to control overfitting

LGBM also has important regularization parameters.

`lambda_l1` and `lambda_l2` specifies L1 or L2 regularization, like XGBoost's `reg_lambda` and `reg_alpha`. The optimal value for these parameters is harder to tune because their magnitude is not directly correlated with overfitting. However, a good search range is (0, 100) for both.

Next, we have `min_gain_to_split`, similar to XGBoost's `gamma`. A conservative search range is (0, 15). It can be used as extra regularization in large parameter grids.

Lastly, we have `bagging_fraction` and `feature_fraction`. `bagging_fraction` takes a value within (0, 1) and specifies the percentage of training samples to be used to train each tree (exactly like `subsample` in XGBoost). To use this parameter, you also need to set `bagging_freq` to an integer value, explanation here.

`feature_fraction` specifies the percentage of features to sample when training each tree. So, it also takes a value between (0, 1).

We have already covered other parameters that affect overfitting (`max_depth`, `num_leaves`, etc.) in earlier sections.

Creating the search grid in Optuna

The optimization process in Optuna requires a function called *objective* that:

- includes the parameter grid to search as a dictionary
- creates a model to try hyperparameter combination sets
- fits the model to the data with a single candidate set
- generates predictions using this model
- scores the predictions based on user-defined metrics and returns it

Here is how it looks like in code:

```python
import optuna  # pip install optuna
from sklearn.metrics import log_loss
from sklearn.model_selection import StratifiedKFold

def objective(trial, X, y):
    param_grid = {}  # to be filled in later
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1121218)

    cv_scores = np.empty(5)
    for idx, (train_idx, test_idx) in enumerate(cv.split(X, y)):
        X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        model = lgbm.LGBMClassifier(objective="binary", **param_grid)
        model.fit(
            X_train,
            y_train,
            eval_set=[(X_test, y_test)],
            eval_metric="binary_logloss",
            early_stopping_rounds=100,
        )
        preds = model.predict_proba(X_test)
        cv_scores[idx] = preds

    return np.mean(cv_scores)
```

8901.py hosted with ♡ by GitHub view raw

Here we have
1. cv
2. loop for fitting lgbm on each fold

In the above `objective` function, we haven't specified the grid yet.

It is optional, but we are performing training inside cross-validation. This ensures that each hyperparameter candidate set gets trained on full data and evaluated more robustly. It also enables us to use early stopping. At the last line, we are returning the mean of the CV scores, which we want to optimize.

Let's focus on creating the grid now. We will include the hyperparameters introduced today with their recommended search ranges:

```python
def objective(trial, X, y):
    param_grid = {
        #         "device_type": trial.suggest_categorical("device_type", ['gpu']),
        "n_estimators": trial.suggest_categorical("n_estimators", [10000]),
        "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.3),
        "num_leaves": trial.suggest_int("num_leaves", 20, 3000, step=20),
        "max_depth": trial.suggest_int("max_depth", 3, 12),
        "min_data_in_leaf": trial.suggest_int("min_data_in_leaf", 200, 10000, step=100),
        "max_bin": trial.suggest_int("max_bin", 200, 300),
        "lambda_l1": trial.suggest_int("lambda_l1", 0, 100, step=5),
        "lambda_l2": trial.suggest_int("lambda_l2", 0, 100, step=5),
        "min_gain_to_split": trial.suggest_float("min_gain_to_split", 0, 15),
        "bagging_fraction": trial.suggest_float(
            "bagging_fraction", 0.2, 0.95, step=0.1
        ),
        "bagging_freq": trial.suggest_categorical("bagging_freq", [1]),
        "feature_fraction": trial.suggest_float(
            "feature_fraction", 0.2, 0.95, step=0.1
        ),
    }

    ...
```

8902.py hosted with ♡ by GitHub view raw

all these values are those ranges above

here we have the parameters grid only

> *If you don't understand the above grid or the `trial` object, check out my article on Optuna.*

Creating Optuna study and running trials

It is time to start the search. Here is the full objective function for reference:

```python
from optuna.integration import LightGBMPruningCallback

def objective(trial, X, y):
    param_grid = {
        # "device_type": trial.suggest_categorical("device_type", ['gpu']),
        "n_estimators": trial.suggest_categorical("n_estimators", [10000]),
        "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.3),
        "num_leaves": trial.suggest_int("num_leaves", 20, 3000, step=20),
        "max_depth": trial.suggest_int("max_depth", 3, 12),
        "min_data_in_leaf": trial.suggest_int("min_data_in_leaf", 200, 10000, step=100),
        "lambda_l1": trial.suggest_int("lambda_l1", 0, 100, step=5),
        "lambda_l2": trial.suggest_int("lambda_l2", 0, 100, step=5),
        "min_gain_to_split": trial.suggest_float("min_gain_to_split", 0, 15),
        "bagging_fraction": trial.suggest_float(
            "bagging_fraction", 0.2, 0.95, step=0.1
        ),
        "bagging_freq": trial.suggest_categorical("bagging_freq", [1]),
        "feature_fraction": trial.suggest_float(
            "feature_fraction", 0.2, 0.95, step=0.1
        ),
    }

    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1121218)

    cv_scores = np.empty(5)
    for idx, (train_idx, test_idx) in enumerate(cv.split(X, y)):
        X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        model = lgbm.LGBMClassifier(objective="binary", **param_grid)
        model.fit(
            X_train,
            y_train,
            eval_set=[(X_test, y_test)],
            eval_metric="binary_logloss",
            early_stopping_rounds=100,
            callbacks=[
                LightGBMPruningCallback(trial, "binary_logloss")
            ],  # Add a pruning callback
        )
        preds = model.predict_proba(X_test)
        cv_scores[idx] = log_loss(y_test, preds)

    return np.mean(cv_scores)
```

8903.py hosted with ♡ by GitHub view raw

both earlier wale combined

To this grid, I also added `LightGBMPruningCallback` from Optuna's `integration` module. This callback class is handy - it can detect unpromising hyperparameter sets before training them on the data, reducing the search time significantly.

You should pass it to LGBM's `fit` method under `callbacks` and set the `trial` object and the evaluation metric you are using as parameters.

Now, let's create the study and run a few trials:

```python
study = optuna.create_study(direction="minimize", study_name="LGBM Classifier")
func = lambda trial: objective(trial, X, y)
study.optimize(func, n_trials=20)
```

8904.py hosted with ♡ by GitHub view raw

After the search is done, call `best_value` and `bast_params` attributes, and you will get an output similar to this:

```
print(f"\tBest value (rmse): {study.best_value:.5f}")
print(f"\tBest params:")

for key, value in study.best_params.items():
    print(f"\t\t{key}: {value}")

---------------------------------------------------
Best value (binary_logloss): 0.35738
        Best params:
                device: gpu
                lambda_l1: 7.71800699380605e-05
                lambda_l2: 4.17890272377219e-06
                bagging_fraction: 0.7000000000000001
                feature_fraction: 0.4
                bagging_freq: 5
                max_depth: 5
                num_leaves: 1007
                min_data_in_leaf: 45
                min_split_gain: 15.703519227860273
                learning_rate: 0.010784015325759629
                n_estimators: 10000
```

8905.py hosted with ♡ by GitHub view raw

Conclusion

That's it! You are now a pro LGBM user. If you implement the things you learned in these two articles, believe me, you are already better than many Kagglers who use LightGBM.

That's because you have a deeper understanding of how the library works, what its parameters represent, and skillfully tune them. This type of fundamental knowledge of a library is always better than rampant code reuse without an ounce of understanding.

To move from *pro* to *master*, I suggest spending some time on the documentation. Thank you for reading!

You might also be interested...