You Are Missing Out on LightGBM. It Crushes XGBoost in Every Aspect

Not anymore, XGBoost, not anymore

Bex T. Sep 2 · 7 min read ⋆

Learn how to crush XGBoost in this comprehensive LightGBM tutorial.

Photo by GR Stocks on Unsplash. All images are by the author unless specified otherwise.

I am confused.

So many people are drawn to XGBoost like a moth to a flame. Yes, it has seen some glorious days in prestigious competitions, and it's *still* the most widely-used ML library.

But, it has been 4 years since XGBoost lost its top spot in terms of performance. In 2017, Microsoft open-sourced LightGBM (Light Gradient Boosting Machine) that gives equally high accuracy with 2–10 times less training speed.

This is a game-changing advantage considering the ubiquity of massive, million-row datasets. There are other distinctions that tip the scales towards LightGBM and give it an edge over XGBoost.

By the end of this post, you will learn about these advantages, including:

- how to develop LightGBM models for classification and regression tasks
- structural differences between XGBoost and LGBM
- how to use early stopping and evaluation sets
- enabling powerful categorical feature support for up 8x times speed increase
- implementing successful cross-validation with LGBM
- hyperparameter tuning with Optuna (Part II)

XGBoost vs. LightGBM

When LGBM got released, it came with ground-breaking changes to the way it grows decision trees.

> *Both XGBoost and LightGBM are ensebmle algorithms. They use a special type of decision trees, also called weak learners, to capture complex, non-linear patterns.*

In XGBoost (and many other libraries), decision trees were built one level at a time:

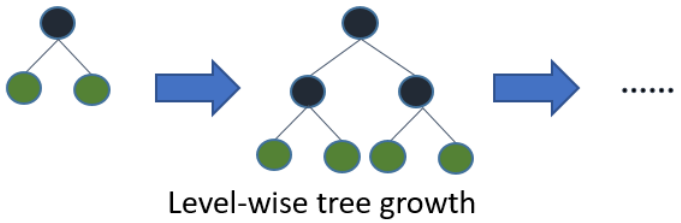

Image from LGBM documentation

This type of structure tends to result in unnecessary nodes and leaves because the trees continued to build until the `max_depth` reached. This led to higher model complexity and training cost runtime.

In contrast, LightGBM takes a leaf-wise approach:

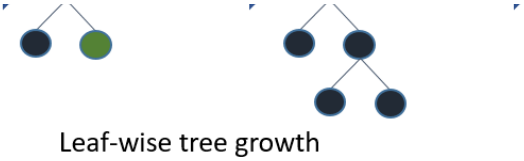

Image from LGBM documentation

The structure continues to grow with the most promising branches and leaves (nodes with the most delta loss), holding the number of the decision leaves constant. (If this doesn't make sense to you, don't sweat. This won't prevent you from effectively using LGBM).

This is one of the main reasons LGBM crushed XGBoost in terms of speed when it first came out.

The following table is the comparison of time cost:

Data	xgboost	xgboost_hist	LightGBM
Higgs	3794.34 s	165.575 s	**130.094 s**
Yahoo LTR	674.322 s	131.462 s	**76.229 s**
MS LTR	1251.27 s	98.386 s	**70.417 s**
Expo	1607.35 s	137.65 s	**62.607 s**
Allstate	2867.22 s	315.256 s	**148.231 s**

LightGBM ran faster than xgboost on all experiment data sets.

Image from LGBM documentation

Above is a benchmark comparison of XGBoost with traditional decision trees and LGBM with leaf-wise structure (first and last columns) on datasets with ~500k-13M samples. It shows that LGBM is orders of magnitude faster than XGB.

LGBM also uses histogram binning of continuous features, which provides even more speed-up than traditional gradient boosting. Binning numeric values significantly decrease the number of split points to consider in decision trees, and they remove the need to use sorting algorithms, which are always computation-heavy.

Inspired by LGBM, XGBoost also introduced histogram-binning, which gave massive speed-up but still not enough to match LGBM's:

Data	xgboost	xgboost_hist	LightGBM
Higgs	3794.34 s	165.575 s	**130.094 s**
Yahoo LTR	674.322 s	131.462 s	**76.229 s**
MS LTR	1251.27 s	98.386 s	**70.417 s**
Expo	1607.35 s	137.65 s	**62.607 s**
Allstate	2867.22 s	315.256 s	**148.231 s**

Image from LGBM documentation**Histogram-binning comparison — second and third columns.**

We will continue exploring the differences in the coming sections.

Model initialization and objectives

Like XGBoost, LGBM has two APIs — core learning API and Sklearn-compatible one. You know I am a big fan of Sklearn, so this tutorial will focus on that version.

> *Sklearn-compatible API of XGBoost and LGBM allows you to integrate their models in the Sklearn ecosystem so that you can use them inside pipelines in combination with other transformers.*

Sklearn API exposes `LGBMRegressor` and `LGBMClassifier`, with the familiar `fit/predict/predict_proba` pattern:

```
import lightgbm as lgbm  # standard alias

clf = lgbm.LGBMClassifier(objective="binary")  # or 'mutliclass'
reg = lgbm.LGBMRegressor()  # default - 'regression'
```

8801.py hosted with ♡ by GitHub view raw

`objective` specifies the type of learning task. Besides the common ones like `binary`, `multiclass` and `regression` tasks, there are others like `poisson`, `tweedie` regressions. See this section of the documentation for the full list of objectives.

```
>>> reg.fit(X, y)

LGBMRegressor()
```

Controlling the number of decision trees

The number of decision trees inside the ensemble significantly affects the results. You can control it using the `n_estimators` parameter in both the classifier and regressor. Below, we will fit an LGBM binary classifier on the Kaggle TPS March dataset with 1000 decision trees:

```
tps_march = pd.read_csv("data/train.csv")
tps_march.head()
```

8802.py hosted with ♡ by GitHub view raw

	id	cat0	cat1	cat2	cat3	cat4	cat5	cat6	cat7	cat8	...	cont2	cont3	cont4	cont5	cont6	cont7	cont8	cont9	cont10	target
0	0	A	I	A	B	B	BI	A	S	Q	...	0.759439	0.795549	0.681917	0.621672	0.592184	0.791921	0.815254	0.965006	0.665915	0
1	1	A	I	A	A	E	BI	K	W	AD	...	0.386385	0.541366	0.388982	0.357778	0.600044	0.408701	0.399353	0.927406	0.493729	0
2	2	A	K	A	A	E	BI	A	E	BM	...	0.343255	0.616352	0.793687	0.552877	0.352113	0.388835	0.412303	0.292696	0.549452	0
3	3	A	K	A	C	E	BI	A	Y	AD	...	0.831147	0.807807	0.800032	0.619147	0.221789	0.897617	0.633669	0.760318	0.934242	0
4	4	A	I	G	B	E	BI	C	G	Q	...	0.338818	0.277308	0.610578	0.128291	0.578764	0.279167	0.351103	0.357084	0.328960	1

5 rows × 32 columns

```
>>> tps_march.shape
(300000, 32)

>>> tps_march.dtypes.value_counts()
object     19
float64    11
int64       2
dtype: int64

>>> X, y = tps_march.drop("target", axis=1), tps_march[["target"]].values.flatten()
```

X, y = tps_march.drop("target", axis=1), tps march[["target"]].values.flatten()

```
11
12  # Encode categoricals
13  >>> X_enc = pd.get_dummies(X)
14
15  >>> clf = lgbm.LGBMClassifier(
16  ...     objective="binary", n_estimators=1000, random_state=1121218
17  ... )
18  >>> clf.fit(X_enc, y)
19  LGBMClassifier(n_estimators=1000, objective='binary', random_state=1121218)
```

8803.py hosted with ♡ by GitHub view raw

Adding more trees leads to more accuracy but increases the risk of overfitting. To combat this, you can create many trees (+2000) and choose a smaller `learning_rate` (more on this later).

```
1  >>> clf = lgbm.LGBMClassifier(
2  ...     objective="binary", n_estimators=3000, learning_rate=0.1, random_state=1121218
3  ...)
4
5  >>> clf.fit(X_enc, y)
6  LGBMClassifier(n_estimators=3000, objective='binary', random_state=1121218)
```

8804.py hosted with ♡ by GitHub view raw

Like in XGBoost, fitting a single decision tree to the data is called a **boosting round**.

Early stopping

Each tree in the ensemble builds on the predictions of the last tree — i.e., each boosting round is an improvement of the last.

If the predictions don't improve after a sequence of rounds, it is sensible to stop the training of the ensemble even if we are not at a hard stop for `n_estimators`.

To achieve this, LGBM provides `early_stopping_rounds` parameter inside the `fit` function. For example, setting it to 100 means we stop the training if the predictions have not improved for the last 100 rounds.

Before looking at a code example, we should learn a couple of concepts connected to early stopping.

Eval sets and metrics

Early stopping is only enabled when you pass a set of evaluation sets to `eval_set` parameter of the `fit` method. These evaluation sets are used to keep track of the quality of the predictions from one boosting round to the next:

```
1  from sklearn.model_selection import tXGBClassifiertest_split
2
3  X_train, X_eval, y_train, y_eval = train_test_split(X_enc, y, test_size=0.1)
4
5  clf = lgbm.LGBMClassifier(objective="binary", n_estimators=10000)
```

```
eval_set = [(X_eval, y_eval)]

clf.fit(
    X_train,
    y_train,
    eval_set=eval_set,
    early_stopping_rounds=100,
    eval_metric="binary_logloss",
)

-------------------------------------------------------

Training until validation scores don't improve for 100 rounds
[1]     valid_0's binary_logloss: 0.542547
[2]     valid_0's binary_logloss: 0.515902
[3]     valid_0's binary_logloss: 0.494678
[4]     valid_0's binary_logloss: 0.477235
[5]     valid_0's binary_logloss: 0.462826
...
[737]   valid_0's binary_logloss: 0.350179
[738]   valid_0's binary_logloss: 0.350162
Early stopping, best iteration is:
[638]   valid_0's binary_logloss: 0.350073
```

8805.py hosted with ♡ by GitHub view raw

In each round of `n_estimators`, a single decision tree is fit to (`X_train`, `y_train`) and predictions are made on the passed evaluation set (`X_eval`, `y_eval`). The quality of predictions is measured with a passed metric in `eval_metric`.

The training stops at the 738th iteration because the validation score has not improved since the 638th one — early stopping of 100 rounds. Now, we have the luxury of creating as many trees as we want and `early_stopping_rounds` can discard the unnecessary ones.

Establish a baseline

Let's establish a baseline score with what we know so far. We will do the same for XGBoost so that we can compare the results:

```
import xgboost as xgb
from sklearn.metrics import log_loss

X, y = tps_march.drop("target", axis=1), tps_march[["target"]].values.flatten()

# Encode categoricals
X_enc = pd.get_dummies(X)
X_train, X_eval, y_train, y_eval = train_test_split(
    X_enc, y, test_size=0.2, stratify=y
  )
```

8806.py hosted with ♡ by GitHub view raw

```
%%time

xgb_clf = xgb.XGBClassifier(
    objective="binary:logistic",
    random_state=1121218,
    n_estimators=10000,
    tree_method="hist",  # enable histogram binning in XGB
)

xgb_clf.fit(
    X_train,
    y_train,
    eval_set=[(X_eval, y_eval)],
```

```
    eval_metric="logloss",
    early_stopping_rounds=150,
    verbose=False,  # Disable logs
)

preds = xgb_clf.predict_proba(X_eval)
print(f"XGBoost logloss on the evaluation set: {log_loss(y_eval, preds):.5f}")

------------------------------------------------------------

XGBoost logloss on the evaluation set: 0.35482
Wall time: 1min 22s
```

8807.py hosted with ♡ by GitHub view raw

```
%%time

lgbm_clf = lgbm.LGBMClassifier(
    objective="binary",
    random_state=1121218,
    n_estimators=10000,
    boosting="gbdt",  # default histogram binning of LGBM
    #     device='gpu'  # uncomment to use GPU training
)

lgbm_clf.fit(
    X_train,
    y_train,
    eval_set=[(X_eval, y_eval)],
    eval_metric="binary_logloss",
    early_stopping_rounds=150,
    verbose=False,  # Disable logs
)

preds = lgbm_clf.predict_proba(X_eval)
print(f"LightGBM logloss on the evaluation set: {log_loss(y_eval, preds):.5f}")

------------------------------------------------

LightGBM logloss on the evaluation set: 0.35256
Wall time: 17.2 s
```

8808.py hosted with ♡ by GitHub view raw

LGBM achieved a smaller loss in ~4 times less runtime. Let's see a final LGBM trick before we move on to cross-validation.

Categorical and missing values support

Histogram binning in LGBM comes with built-in support for handling missing values and categorical features. TPS March dataset contains 19 categoricals, and we have been using one-hot encoding up to this point.

This time, we will let LGBM deal with categoricals and compare the results with XGBoost once again:

```
X, y = tps_march.drop("target", axis=1), tps_march[["target"]].values.flatten()

# Extract categoricals and their indices
cat_features = X.select_dtypes(exclude=np.number).columns.to_list()
cat_idx = [X.columns.get_loc(col) for col in cat_features]

# Convert cat_features to pd.Categorical dtype
for col in cat_features:
    X[col] = pd.Categorical(X[col])

# Unencoded train/test sets
X_train, X_eval, y_train, y_eval = train_test_split(
```

```
    X, y, test_size=0.2, random_state=4, stratify=y
)
```

8809.py hosted with ♡ by GitHub view raw

To specify the categorical features, pass a list of their indices to `categorical_feature` parameter in the `fit` method:

```
%%time

# Model initialization is the same
eval_set = [(X_eval, y_eval)]

_ = lgbm_clf.fit(
    X_train,
    y_train,
    categorical_feature=cat_idx,  # Specify the categoricals
    eval_set=eval_set,
    early_stopping_rounds=150,
    eval_metric="logloss",
    verbose=False,
)

preds = lgbm_clf.predict_proba(X_eval)
loss = log_loss(y_eval, preds)
print(f"LGBM logloss with default cateogircal feature handling: {loss:.5f}")

----------------------------------------------------------------

LGBM logloss with default cateogircal feature handling: 0.34823
Wall time: 15 s
```

8810.py hosted with ♡ by GitHub view raw

> *You can achieve up to 8x speed up if you use `pandas.Categorical` data type when using LGBM.*

	XGB (one-hot)	LGBM (one-hot)	LGBM (cat-handling)
score	0.35482	0.35256	0.34823
runtime	1min 22s	17.2 s	15 s

The table shows the final scores and runtimes of both models. As you can see, the version with default categorical handling beat others both in accuracy and speed. Cheers!

Cross-validation with LightGBM

The most common way of doing CV with LGBM is to use Sklearn CV splitters.

I am not talking about utility functions like `cross_validate` or `cross_val_score` but splitters like `KFold` or `StratifiedKFold` with their `split` method. Doing CV in this way gives you more control over the whole process.

> *I have talked many times about the importance of cross-validation. You can read this post for more details.*

Also, it enables you to use early stopping during cross-validation in a hassle-free manner. Here is what this looks like for the TPS March data:

```python
import time

from sklearn.model_selection import StratifiedKFold

N_SPLITS = 7
strat_kf = StratifiedKFold(n_splits=N_SPLITS, shuffle=True, random_state=1121218)

scores = np.empty(N_SPLITS)
for idx, (train_idx, test_idx) in enumerate(strat_kf.split(X, y)):
    print("=" * 12 + f"Training fold {idx}" + 12 * "=")
    start = time.time()

    X_train, X_val = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_val = y[train_idx], y[test_idx]
    eval_set = [(X_val, y_val)]

    lgbm_clf = lgbm.LGBMClassifier(n_estimators=10000)
    lgbm_clf.fit(
        X_train,
        y_train,
        eval_set=eval_set,
        categorical_feature=cat_idx,
        early_stopping_rounds=200,
        eval_metric="binary_logloss",
        verbose=False,
    )

    preds = lgbm_clf.predict_proba(X_val)
    loss = log_loss(y_val, preds)
    scores[idx] = loss
    runtime = time.time() - start
    print(f"Fold {idx} finished with score: {loss:.5f} in {runtime:.2f} seconds.\n")

------------------------------------------------------------------

============Training fold 0============
Fold 0 finished with score: 0.34547 in 15.46 seconds.

============Training fold 1============
Fold 1 finished with score: 0.34824 in 14.90 seconds.

============Training fold 2============
Fold 2 finished with score: 0.34665 in 12.64 seconds.

============Training fold 3============
Fold 3 finished with score: 0.35015 in 14.78 seconds.

============Training fold 4============
Fold 4 finished with score: 0.34617 in 14.61 seconds.

============Training fold 5============
Fold 5 finished with score: 0.34833 in 13.24 seconds.

============Training fold 6============
Fold 6 finished with score: 0.35155 in 13.27 seconds.
print(f"Average log loss: {np.mean(scores):.4f}")
Average log loss: 0.3481
```

8811.py hosted with ♡ by GitHub view raw

First, create a CV splitter — we are choosing `StratifiedKFold` because it is a classification problem. Then, loop through each train/test sets using `split`. In each fold, initialize and train a new LGBM model and optionally report the score and runtime. That's it! That's how most people do CV, including on Kaggle.

Conclusion

In this post, we learned pure modeling techniques with LightGBM. Next up, we will explore how to squeeze every bit of performance out of LGBM

models using Optuna.

Specifically, Part II of this article will include a detailed overview of the most important LGBM hyperparameters and introduce a well-tested hyperparameter tuning workflow. It is already out — read it here.

You might also be interested...

25 NumPy Functions You Never Knew Existed | P (Guarantee = 0.85)
Become a NumPy Ninja
towardsdatascience.com

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

Tired of Cliché Datasets? Here are 18 Awesome Alternatives From All Domains
Edit description
towardsdatascience.com

Love 3Blue1Brown Animations? Learn How to Create Your Own in Python in 10 Minutes
Edit description
towardsdatascience.com

7 Cool Python Packages Kagglers Are Using Without Telling You
Edit description
towardsdatascience.com