LightGBM Documentation

Release

Microsoft Corporation

Oct 09, 2017

Contents:

CHAPTER 1

Installation Guide

Here is the guide for the build of CLI version.

For the build of Python-package and R-package, please refer to Python-package and R-package folders respectively.

Windows

LightGBM can use Visual Studio, MSBuild with CMake or MinGW to build in Windows.

Visual Studio (or MSBuild)

With GUI

1. Install Visual Studio.
2. Download zip archive and unzip it.
3. Go to `LightGBM-master/windows` folder.
4. Open `LightGBM.sln` file with Visual Studio, choose `Release` configuration and click `BUILD-> Build Solution (Ctrl+Shift+B)`.

   If you have errors about **Platform Toolset**, go to `PROJECT-> Properties-> Configuration Properties-> General` and select the toolset installed on your machine.

The exe file will be in `LightGBM-master/windows/x64/Release` folder.

From Command Line

1. Install Git for Windows, CMake (3.8 or higher) and MSBuild (MSbuild is not needed if **Visual Studio** is installed).
2. Run the following commands:

```
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -DCMAKE_GENERATOR_PLATFORM=x64 ..
cmake --build . --target ALL_BUILD --config Release
```

The exe and dll files will be in `LightGBM/Release` folder.

MinGW64

1. Install Git for Windows, CMake and MinGW-w64.
2. Run the following commands:

```
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
mingw32-make.exe -j4
```

The exe and dll files will be in `LightGBM/` folder.

Note: you may need to run the `cmake -G "MinGW Makefiles" ..` one more time if met `sh.exe was found in your PATH` error.

Linux

LightGBM uses `CMake` to build. Run the following commands:

```
git clone --recursive https://github.com/Microsoft/LightGBM ; cd LightGBM
mkdir build ; cd build
cmake ..
make -j4
```

Note: glibc >= 2.14 is required.

OSX

LightGBM depends on **OpenMP** for compiling, which isn't supported by Apple Clang.

Please install **gcc/g++** by using the following commands:

```
brew install cmake
brew install gcc --without-multilib
```

Then install LightGBM:

```
git clone --recursive https://github.com/Microsoft/LightGBM ; cd LightGBM
export CXX=g++-7 CC=gcc-7
mkdir build ; cd build
```

```
cmake ..
make -j4
```

Docker

Refer to Docker folder.

Build MPI Version

The default build version of LightGBM is based on socket. LightGBM also supports MPI. MPI is a high performance communication approach with RDMA support.

If you need to run a parallel learning application with high performance communication, you can build the LightGBM with MPI support.

Windows

With GUI

1. You need to install MS MPI first. Both `msmpisdk.msi` and `MSMpiSetup.exe` are needed.
2. Install Visual Studio.
3. Download zip archive and unzip it.
4. Go to `LightGBM-master/windows` folder.
4. Open `LightGBM.sln` file with Visual Studio, choose `Release_mpi` configuration and click `BUILD-> Build Solution (Ctrl+Shift+B)`.

   If you have errors about **Platform Toolset**, go to `PROJECT-> Properties-> Configuration Properties-> General` and select the toolset installed on your machine.

The exe file will be in `LightGBM-master/windows/x64/Release_mpi` folder.

From Command Line

1. You need to install MS MPI first. Both `msmpisdk.msi` and `MSMpiSetup.exe` are needed.
2. Install Git for Windows, CMake (3.8 or higher) and MSBuild (MSbuild is not needed if **Visual Studio** is installed).
3. Run the following commands:

```
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -DCMAKE_GENERATOR_PLATFORM=x64 -DUSE_MPI=ON ..
cmake --build . --target ALL_BUILD --config Release
```

The exe and dll files will be in `LightGBM/Release` folder.

Note: Build MPI version by **MinGW** is not supported due to the miss of MPI library in it.

Linux

You need to install Open MPI first.

Then run the following commands:

```
git clone --recursive https://github.com/Microsoft/LightGBM ; cd LightGBM
mkdir build ; cd build
cmake -DUSE_MPI=ON ..
make -j4
```

Note: glibc >= 2.14 is required.

OSX

Install **gcc** and **Open MPI** first:

```
brew install openmpi
brew install cmake
brew install gcc --without-multilib
```

Then run the following commands:

```
git clone --recursive https://github.com/Microsoft/LightGBM ; cd LightGBM
export CXX=g++-7 CC=gcc-7
mkdir build ; cd build
cmake -DUSE_MPI=ON ..
make -j4
```

Build GPU Version

Linux

The following dependencies should be installed before compilation:

- OpenCL 1.2 headers and libraries, which is usually provided by GPU manufacture.

  The generic OpenCL ICD packages (for example, Debian package `cl-icd-libopencl1` and `cl-icd-opencl-dev`) can also be used.

- libboost 1.56 or later (1.61 or later recommended).

  We use Boost.Compute as the interface to GPU, which is part of the Boost library since version 1.61. However, since we include the source code of Boost.Compute as a submodule, we only require the host has Boost 1.56 or later installed. We also use Boost.Align for memory allocation. Boost.Compute requires Boost.System and Boost.Filesystem to store offline kernel cache.

  The following Debian packages should provide necessary Boost libraries: `libboost-dev`, `libboost-system-dev`, `libboost-filesystem-dev`.

- CMake 3.2 or later.

To build LightGBM GPU version, run the following commands:

```
git clone --recursive https://github.com/Microsoft/LightGBM ; cd LightGBM
mkdir build ; cd build
cmake -DUSE_GPU=1 ..
make -j4
```

Windows

If you use **MinGW**, the build procedure are similar to the build in Linux. Refer to GPU Windows Compilation to get more details.

Following procedure is for the MSVC(Microsoft Visual C++) build.

1. Install Git for Windows, CMake (3.8 or higher) and MSBuild (MSbuild is not needed if **Visual Studio** is installed).
2. Install **OpenCL** for Windows. The installation depends on the brand (NVIDIA, AMD, Intel) of your GPU card.
   - For running on Intel, get Intel SDK for OpenCL.
   - For running on AMD, get AMD APP SDK.
   - For running on NVIDIA, get CUDA Toolkit.
3. Install Boost Binary.

   Note: match your Visual C++ version:

   Visual Studio 2013 -> `msvc-12.0-64.exe`,

   Visual Studio 2015 -> `msvc-14.0-64.exe`,

   Visual Studio 2017 -> `msvc-14.1-64.exe`.
4. Run the following commands:

   ```
   Set BOOST_ROOT=C:\local\boost_1_64_0\
   Set BOOST_LIBRARYDIR=C:\local\boost_1_64_0\lib64-msvc-14.0
   git clone --recursive https://github.com/Microsoft/LightGBM
   cd LightGBM
   mkdir build
   cd build
   cmake -DCMAKE_GENERATOR_PLATFORM=x64 -DUSE_GPU=1 ..
   cmake --build . --target ALL_BUILD --config Release
   ```

   Note: `C:\local\boost_1_64_0\` and `C:\local\boost_1_64_0\lib64-msvc-14.0` are locations of your Boost binaries. You also can set them to the environment variable to avoid `Set ...` commands when build.

Docker

Refer to GPU Docker folder.

CHAPTER 2

Quick Start

This is a quick start guide for LightGBM of cli version.

Follow the *Installation Guide* to install LightGBM first.

List of other Helpful Links

- *Parameters*
- *Parameters Tuning*
- *Python-package Quick Start*
- *Python API*

Training Data Format

LightGBM supports input data file with CSV, TSV and LibSVM formats.

Label is the data of first column, and there is no header in the file.

Categorical Feature Support

update 12/5/2016:

LightGBM can use categorical feature directly (without one-hot coding). The experiment on Expo data shows about 8x speed-up compared with one-hot coding.

For the setting details, please refer to *Parameters*.

Weight and Query/Group Data

LightGBM also support weighted training, it needs an additional *weight data*. And it needs an additional *query data* for ranking task.

update 11/3/2016:

1. support input with header now
2. can specific label column, weight column and query/group id column. Both index and column are supported
3. can specific a list of ignored columns

Parameter Quick Look

The parameter format is `key1=value1 key2=value2 ....` And parameters can be in both config file and command line.

Some important parameters:

- `config`, default=`""`, type=string, alias=`config_file`
  - path of config file
- `task`, default=`train`, type=enum, options=`train`,`prediction`
  - `train` for training
  - `prediction` for prediction.
- `application`, default=`regression`, type=enum, options=`regression`,`regression_l1`,`huber`,`fair`,`poisson`,`bin` alias=`objective`,`app`
  - `regression`, regression application
    - `regression_l2`, L2 loss, alias=`mean_squared_error`,`mse`
    - `regression_l1`, L1 loss, alias=`mean_absolute_error`,`mae`
    - `huber`, Huber loss
    - `fair`, Fair loss
    - `poisson`, Poisson regression
  - `binary`, binary classification application
  - `lambdarank`, lambdarank application
    - The label should be `int` type in lambdarank tasks, and larger number represent the higher relevance (e.g. 0:bad, 1:fair, 2:good, 3:perfect).
    - `label_gain` can be used to set the gain(weight) of `int` label.
  - `multiclass`, multi-class classification application, should set `num_class` as well
- `boosting`, default=`gbdt`, type=enum, options=`gbdt`,`rf`,`dart`,`goss`, alias=`boost`,`boosting_type`
  - `gbdt`, traditional Gradient Boosting Decision Tree
  - `rf`, Random Forest
  - `dart`, Dropouts meet Multiple Additive Regression Trees
  - `goss`, Gradient-based One-Side Sampling
- `data`, default=`""`, type=string, alias=`train`,`train_data`
  - training data, LightGBM will train from this data
- `valid`, default=`""`, type=multi-string, alias=`test`,`valid_data`,`test_data`

  - validation/test data, LightGBM will output metrics for these data
  - support multi validation data, separate by `,`
- `num_iterations`, default=`100`, type=int, alias=`num_iteration`,`num_tree`,`num_trees`,`num_round`,`num_rounds`
  - number of boosting iterations/trees
- `learning_rate`, default=`0.1`, type=double, alias=`shrinkage_rate`
  - shrinkage rate
- `num_leaves`, default=`31`, type=int, alias=`num_leaf`
  - number of leaves in one tree
- `tree_learner`, default=`serial`, type=enum, options=`serial`,`feature`,`data`
  - `serial`, single machine tree learner
  - `feature`, feature parallel tree learner
  - `data`, data parallel tree learner
  - Refer to *Parallel Learning Guide* to get more details.
- `num_threads`, default=OpenMP_default, type=int, alias=`num_thread`,`nthread`
  - Number of threads for LightGBM.
  - For the best speed, set this to the number of **real CPU cores**, not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core).
  - For parallel learning, should not use full CPU cores since this will cause poor performance for the network.
- `max_depth`, default=`-1`, type=int
  - Limit the max depth for tree model. This is used to deal with overfit when #data is small. Tree still grow by leaf-wise.
  - `< 0` means no limit
- `min_data_in_leaf`, default=`20`, type=int, alias=`min_data_per_leaf` , `min_data`
  - Minimal number of data in one leaf. Can use this to deal with over-fit.
- `min_sum_hessian_in_leaf`, default=`1e-3`, type=double, alias=`min_sum_hessian_per_leaf`, `min_sum_hessian`, `min_hessian`
  - Minimal sum hessian in one leaf. Like `min_data_in_leaf`, can use this to deal with over-fit.

For all parameters, please refer to *Parameters*.

Run LightGBM

For Windows:

```
lightgbm.exe config=your_config_file other_args ...
```

For Unix:

```
./lightgbm config=your_config_file other_args ...
```

Parameters can be both in the config file and command line, and the parameters in command line have higher priority than in config file. For example, following command line will keep 'num_trees=10' and ignore same parameter in config file.

```
./lightgbm config=train.conf num_trees=10
```

Examples

- Binary Classification
- Regression
- Lambdarank
- Parallel Learning

CHAPTER 3

Python Package Introduction

This document gives a basic walkthrough of LightGBM Python-package.

List of other Helpful Links

- Python Examples
- *Python API*
- *Parameters Tuning*

Install

Install Python-package dependencies, `setuptools`, `wheel`, `numpy` and `scipy` are required, `scikit-learn` is required for sklearn interface and recommended:

```
pip install setuptools wheel numpy scipy scikit-learn -U
```

Refer to Python-package folder for the installation guide.

To verify your installation, try to `import lightgbm` in Python:

```
import lightgbm as lgb
```

Data Interface

The LightGBM Python module is able to load data from:

- libsvm/tsv/csv txt format file
- Numpy 2D array, pandas object
- LightGBM binary file

The data is stored in a `Dataset` object.

To load a libsvm text file or a LightGBM binary file into `Dataset`:

```
train_data = lgb.Dataset('train.svm.bin')
```

To load a numpy array into `Dataset`:

```
data = np.random.rand(500, 10) # 500 entities, each contains 10 features
label = np.random.randint(2, size=500) # binary target
train_data = lgb.Dataset(data, label=label)
```

To load a scpiy.sparse.csr_matrix array into `Dataset`:

```
csr = scipy.sparse.csr_matrix((dat, (row, col)))
train_data = lgb.Dataset(csr)
```

Saving `Dataset` into a LightGBM binary file will make loading faster:

```
train_data = lgb.Dataset('train.svm.txt')
train_data.save_binary('train.bin')
```

Create validation data:

```
test_data = train_data.create_valid('test.svm')
```

or

```
test_data = lgb.Dataset('test.svm', reference=train_data)
```

In LightGBM, the validation data should be aligned with training data.

Specific feature names and categorical features:

```
train_data = lgb.Dataset(data, label=label, feature_name=['c1', 'c2', 'c3'],␣
↪categorical_feature=['c3'])
```

LightGBM can use categorical features as input directly. It doesn't need to covert to one-hot coding, and is much faster than one-hot coding (about 8x speed-up).

Note:You should convert your categorical features to int type before you construct `Dataset`.

Weights can be set when needed:

```
w = np.random.rand(500, )
train_data = lgb.Dataset(data, label=label, weight=w)
```

or

```
train_data = lgb.Dataset(data, label=label)
w = np.random.rand(500, )
train_data.set_weight(w)
```

And you can use `Dataset.set_init_score()` to set initial score, and `Dataset.set_group()` to set group/query data for ranking tasks.

Memory efficent usage

The `Dataset` object in LightGBM is very memory-efficient, due to it only need to save discrete bins. However, Numpy/Array/Pandas object is memory cost. If you concern about your memory consumption. You can save memory accroding to following:

1. Let `free_raw_data=True`(default is `True`) when constructing the `Dataset`
2. Explicit set `raw_data=None` after the `Dataset` has been constructed
3. Call `gc`

Setting Parameters

LightGBM can use either a list of pairs or a dictionary to set *Parameters*. For instance:

- Booster parameters:

```
param = {'num_leaves':31, 'num_trees':100, 'objective':'binary'}
param['metric'] = 'auc'
```

- You can also specify multiple eval metrics:

```
param['metric'] = ['auc', 'binary_logloss']
```

Training

Training a model requires a parameter list and data set.

```
num_round = 10
bst = lgb.train(param, train_data, num_round, valid_sets=[test_data])
```

After training, the model can be saved.

```
bst.save_model('model.txt')
```

The trained model can also be dumped to JSON format.

```
# dump model
json_model = bst.dump_model()
```

A saved model can be loaded.

```
bst = lgb.Booster(model_file='model.txt') #init model
```

CV

Training with 5-fold CV:

```
num_round = 10
lgb.cv(param, train_data, num_round, nfold=5)
```

Early Stopping

If you have a validation set, you can use early stopping to find the optimal number of boosting rounds. Early stopping requires at least one set in `valid_sets`. If there's more than one, it will use all of them.

```
bst = lgb.train(param, train_data, num_round, valid_sets=valid_sets, early_stopping_
↪rounds=10)
bst.save_model('model.txt', num_iteration=bst.best_iteration)
```

The model will train until the validation score stops improving. Validation error needs to improve at least every `early_stopping_rounds` to continue training.

If early stopping occurs, the model will have an additional field: `bst.best_iteration`. Note that `train()` will return a model from the last iteration, not the best one. And you can set `num_iteration=bst.best_iteration` when saving model.

This works with both metrics to minimize (L2, log loss, etc.) and to maximize (NDCG, AUC). Note that if you specify more than one evaluation metric, all of them will be used for early stopping.

Prediction

A model that has been trained or loaded can perform predictions on data sets.

```
# 7 entities, each contains 10 features
data = np.random.rand(7, 10)
ypred = bst.predict(data)
```

If early stopping is enabled during training, you can get predictions from the best iteration with `bst.best_iteration`:

```
ypred = bst.predict(data, num_iteration=bst.best_iteration)
```

CHAPTER 4

Features

This is a short introduction for the features and algorithms used in LightGBM.

This page doesn't contain detailed algorithms, please refer to cited papers or source code if you are interested.

Optimization in Speed and Memory Usage

Many boosting tools use pre-sorted based algorithms*[1, 2]* (e.g. default algorithm in xgboost) for decision tree learning. It is a simple solution, but not easy to optimize.

LightGBM uses the histogram based algorithms*[3, 4, 5]*, which bucketing continuous feature(attribute) values into discrete bins, to speed up training procedure and reduce memory usage. Following are advantages for histogram based algorithms:

- **Reduce calculation cost of split gain**
  - Pre-sorted based algorithms need `O(#data)` times calculation
  - Histogram based algorithms only need to calculate `O(#bins)` times, and `#bins` is far smaller than `#data`
    - It still needs `O(#data)` times to construct histogram, which only contain sum-up operation
- **Use histogram subtraction for further speed-up**
  - To get one leaf's histograms in a binary tree, can use the histogram subtraction of its parent and its neighbor
  - So it only need to construct histograms for one leaf (with smaller `#data` than its neighbor), then can get histograms of its neighbor by histogram subtraction with small cost(`O(#bins)`)
- **Reduce memory usage**
  - Can replace continuous values to discrete bins. If `#bins` is small, can use small data type, e.g. uint8_t, to store training data
  - No need to store additional information for pre-sorting feature values
- **Reduce communication cost for parallel learning**

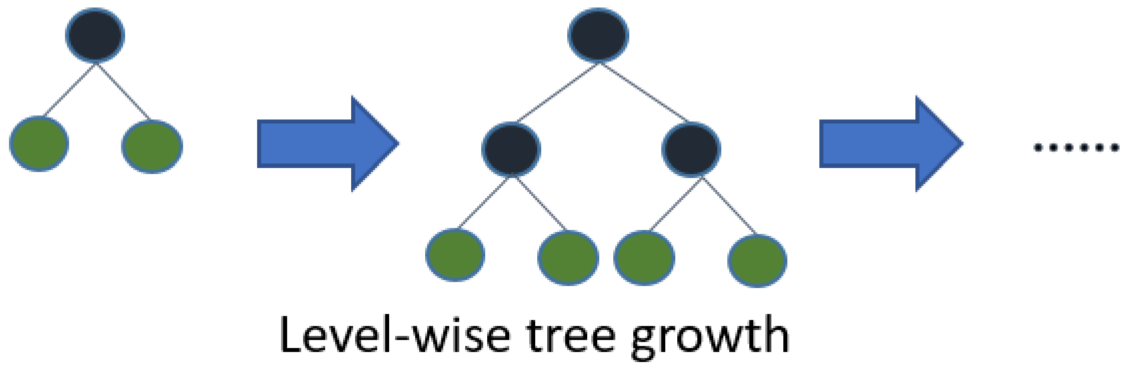
Sparse Optimization

- Only need `O(2 * #non_zero_data)` to construct histogram for sparse features

Optimization in Accuracy

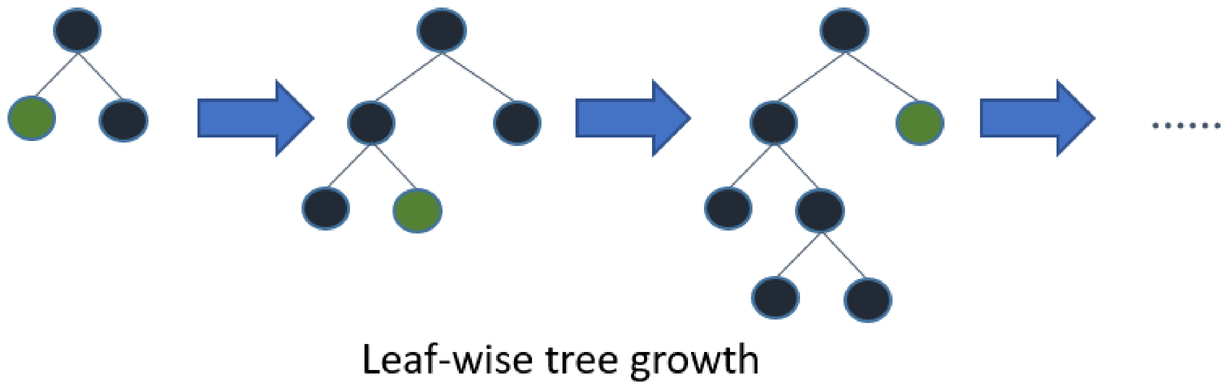
Leaf-wise (Best-first) Tree Growth

Most decision tree learning algorithms grow tree by level(depth)-wise, like the following image:



LightGBM grows tree by leaf-wise(best-first)*[6]*. It will choose the leaf with max delta loss to grow. When growing same `#leaf`, leaf-wise algorithm can reduce more loss than level-wise algorithm.

Leaf-wise may cause over-fitting when `#data` is small. So, LightGBM can use an additional parameter `max_depth` to limit depth of tree and avoid over-fitting (tree still grows by leaf-wise).



Optimal Split for Categorical Features

We often convert the categorical features into one-hot coding. However, it is not a good solution in tree learner. The reason is, for the high cardinality categorical features, it will grow the very unbalance tree, and needs to grow very deep to achieve the good accuracy.

Actually, the optimal solution is partitioning the categorical feature into 2 subsets, and there are `2^(k-1) - 1` possible partitions. But there is a efficient solution for regression tree*[7]*. It needs about `k * log(k)` to find the optimal partition.

The basic idea is reordering the categories according to the relevance of training target. More specifically, reordering the histogram (of categorical feature) according to it's accumulate values (`sum_gradient / sum_hessian`), then find the best split on the sorted histogram.

Optimization in Network Communication

It only needs to use some collective communication algorithms, like "All reduce", "All gather" and "Reduce scatter", in parallel learning of LightGBM. LightGBM implement state-of-art algorithms*[8]*. These collective communication algorithms can provide much better performance than point-to-point communication.

Optimization in Parallel Learning

LightGBM provides following parallel learning algorithms.

Feature Parallel

Traditional Algorithm

Feature parallel aims to parallel the "Find Best Split" in the decision tree. The procedure of traditional feature parallel is:

1. Partition data vertically (different machines have different feature set)
2. Workers find local best split point {feature, threshold} on local feature set
3. Communicate local best splits with each other and get the best one
4. Worker with best split to perform split, then send the split result of data to other workers
5. Other workers split data according received data

The shortage of traditional feature parallel:

- Has computation overhead, since it cannot speed up "split", whose time complexity is `O(#data)`. Thus, feature parallel cannot speed up well when `#data` is large.
- Need communication of split result, which cost about `O(#data / 8)` (one bit for one data).

Feature Parallel in LightGBM

Since feature parallel cannot speed up well when `#data` is large, we make a little change here: instead of partitioning data vertically, every worker holds the full data. Thus, LightGBM doesn't need to communicate for split result of data since every worker know how to split data. And `#data` won't be larger, so it is reasonable to hold full data in every machine.

The procedure of feature parallel in LightGBM:

1. Workers find local best split point{feature, threshold} on local feature set
2. Communicate local best splits with each other and get the best one

3. Perform best split

However, this feature parallel algorithm still suffers from computation overhead for "split" when `#data` is large. So it will be better to use data parallel when `#data` is large.

Data Parallel

Traditional Algorithm

Data parallel aims to parallel the whole decision learning. The procedure of data parallel is:

1. Partition data horizontally
2. Workers use local data to construct local histograms
3. Merge global histograms from all local histograms
4. Find best split from merged global histograms, then perform splits

The shortage of traditional data parallel:

- High communication cost. If using point-to-point communication algorithm, communication cost for one machine is about `O(#machine * #feature * #bin)`. If using collective communication algorithm (e.g. "All Reduce"), communication cost is about `O(2 * #feature * #bin)` (check cost of "All Reduce" in chapter 4.5 at *[8]*).

Data Parallel in LightGBM

We reduce communication cost of data parallel in LightGBM:

1. Instead of "Merge global histograms from all local histograms", LightGBM use "Reduce Scatter" to merge histograms of different(non-overlapping) features for different workers. Then workers find local best split on local merged histograms and sync up global best split.
2. As aforementioned, LightGBM use histogram subtraction to speed up training. Based on this, we can communicate histograms only for one leaf, and get its neighbor's histograms by subtraction as well.

Above all, we reduce communication cost to `O(0.5 * #feature * #bin)` for data parallel in LightGBM.

Voting Parallel

Voting parallel further reduce the communication cost in *Data Parallel* to constant cost. It uses two stage voting to reduce the communication cost of feature histograms*[9]*.

GPU Support

Thanks @huanzhang12 for contributing this feature. Please read*[10]* to get more details.

- *GPU Installation*
- *GPU Tutorial*

Applications and Metrics

Support following application:

- regression, the objective function is L2 loss
- binary classification, the objective function is logloss
- multi classification
- lambdarank, the objective function is lambdarank with NDCG

Support following metrics:

- L1 loss
- L2 loss
- Log loss
- Classification error rate
- AUC
- NDCG
- Multi class log loss
- Multi class error rate

For more details, please refer to *Parameters*.

Other Features

- Limit `max_depth` of tree while grows tree leaf-wise
- DART
- L1/L2 regularization
- Bagging
- Column(feature) sub-sample
- Continued train with input GBDT model
- Continued train with the input score file
- Weighted training
- Validation metric output during training
- Multi validation data
- Multi metrics
- Early stopping (both training and prediction)
- Prediction for leaf index

For more details, please refer to *Parameters*.

References

[1] Mehta, Manish, Rakesh Agrawal, and Jorma Rissanen. "SLIQ: A fast scalable classifier for data mining." International Conference on Extending Database Technology. Springer Berlin Heidelberg, 1996.

[2] Shafer, John, Rakesh Agrawal, and Manish Mehta. "SPRINT: A scalable parallel classifier for data mining." Proc. 1996 Int. Conf. Very Large Data Bases. 1996.

[3] Ranka, Sanjay, and V. Singh. "CLOUDS: A decision tree classifier for large datasets." Proceedings of the 4th Knowledge Discovery and Data Mining Conference. 1998.

[4] Machado, F. P. "Communication and memory efficient parallel decision tree construction." (2003).

[5] Li, Ping, Qiang Wu, and Christopher J. Burges. "Mcrank: Learning to rank using multiple classification and gradient boosting." Advances in neural information processing systems. 2007.

[6] Shi, Haijian. "Best-first decision tree learning." Diss. The University of Waikato, 2007.

[7] Walter D. Fisher. "On Grouping for Maximum Homogeneity." Journal of the American Statistical Association. Vol. 53, No. 284 (Dec., 1958), pp. 789-798.

[8] Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH." International Journal of High Performance Computing Applications 19.1 (2005): 49-66.

[9] Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, Tieyan Liu. "A Communication-Efficient Parallel Algorithm for Decision Tree." Advances in Neural Information Processing Systems 29 (NIPS 2016).

[10] Huan Zhang, Si Si and Cho-Jui Hsieh. "GPU Acceleration for Large-scale Tree Boosting." arXiv:1706.08359, 2017.

CHAPTER 5

Experiments

Comparison Experiment

For the detailed experiment scripts and output logs, please refer to this repo.

Data

We use 4 datasets to conduct our comparison experiments. Details of data are listed in the following table:

Data	Task	Link	#Train_Set	#Feature	Comments
Higgs	Binary classification	link	10,500,000	28	use last 500,000 samples as test set
Yahoo LTR	Learning to rank	link	473,134	700	set1.train as train, set1.test as test
MS LTR	Learning to rank	link	2,270,296	137	{S1,S2,S3} as train set, {S5} as test set
Expo	Binary classification	link	11,000,000	700	use last 1,000,000 as test set
Allstate	Binary classification	link	13,184,290	4228	use last 1,000,000 as test set

Environment

We use one Linux server as experiment platform, details are listed in the following table:

OS	CPU	Memory
Ubuntu 14.04 LTS	2 * E5-2670 v3	DDR4 2133Mhz, 256GB

Baseline

We use xgboost as a baseline.

Both xgboost and LightGBM are built with OpenMP support.

Settings

We set up total 3 settings for experiments, the parameters of these settings are:

1. xgboost:

```
eta = 0.1
max_depth = 8
num_round = 500
nthread = 16
tree_method = exact
min_child_weight = 100
```

2. xgboost_hist (using histogram based algorithm):

```
eta = 0.1
num_round = 500
nthread = 16
tree_method = approx
min_child_weight = 100
tree_method = hist
grow_policy = lossguide
max_depth = 0
max_leaves = 255
```

3. LightGBM:

```
learning_rate = 0.1
num_leaves = 255
num_trees = 500
num_threads = 16
min_data_in_leaf = 0
min_sum_hessian_in_leaf = 100
```

xgboost grows tree depth-wise and controls model complexity by `max_depth`. LightGBM uses leaf-wise algorithm instead and controls model complexity by `num_leaves`. So we cannot compare them in the exact same model setting. For the tradeoff, we use xgboost with `max_depth=8`, which will have max number leaves to 255, to compare with LightGBM with `num_leves=255`.

Other parameters are default values.

Result

Speed

For speed comparison, we only run the training task, which is without any test or metric output. And we don't count the time for IO.

The following table is the comparison of time cost:

Data	xgboost	xgboost_hist	LightGBM
Higgs	3794.34 s	551.898 s	**238.505513 s**
Yahoo LTR	674.322 s	265.302 s	**150.18644 s**
MS LTR	1251.27 s	385.201 s	**215.320316 s**
Expo	1607.35 s	588.253 s	**138.504179 s**
Allstate	2867.22 s	1355.71 s	**348.084475 s**

We found LightGBM is faster than xgboost on all experiment data sets.

Accuracy

For accuracy comparison, we use the accuracy on test data set to have a fair comparison.

Data	Metric	xgboost	xgboost_hist	LightGBM
Higgs	AUC	0.839593	0.845605	0.845154
Yahoo LTR	$NDCG_1$	0.719748	0.720223	0.732466
	$NDCG_3$	0.717813	0.721519	0.738048
	$NDCG_5$	0.737849	0.739904	0.756548
	$NDCG_{10}$	0.78089	0.783013	0.796818
MS LTR	$NDCG_1$	0.483956	0.488649	0.524255
	$NDCG_3$	0.467951	0.473184	0.505327
	$NDCG_5$	0.472476	0.477438	0.510007
	$NDCG_{10}$	0.492429	0.496967	0.527371
Expo	AUC	0.756713	0.777777	0.777543
Allstate	AUC	0.607201	0.609042	0.609167

Memory Consumption

We monitor RES while running training task. And we set `two_round=true` (will increase data-loading time, but reduce peak memory usage, not affect training speed or accuracy) in LightGBM to reduce peak memory usage.

Data	xgboost	xgboost_hist	LightGBM
Higgs	4.853GB	3.784GB	**0.868GB**
Yahoo LTR	1.907GB	1.468GB	**0.831GB**
MS LTR	5.469GB	3.654GB	**0.886GB**
Expo	1.553GB	1.393GB	**0.543GB**
Allstate	6.237GB	4.990GB	**1.027GB**

Parallel Experiment

Data

We use a terabyte click log dataset to conduct parallel experiments. Details are listed in following table:

Data	Task	Link	#Data	#Feature
Criteo	Binary classification	link	1,700,000,000	67

This data contains 13 integer features and 26 category features of 24 days click log. We statistic the CTR and count for these 26 category features from the first ten days, then use next ten days' data, which had been replaced the category features by the corresponding CTR and count, as training data. The processed training data hava total 1.7 billions records and 67 features.

Environment

We use 16 Windows servers as experiment platform, details are listed in following table:

OS	CPU	Memory	Network Adapter
Windows Server 2012	2 * E5-2670 v2	DDR3 1600Mhz, 256GB	Mellanox ConnectX-3, 54Gbps, RDMA support

Settings

```
learning_rate = 0.1
num_leaves = 255
num_trees = 100
num_thread = 16
tree_learner = data
```

We use data parallel here, since this data is large in `#data` but small in `#feature`.

Other parameters are default values.

Result

#Machine	Time per Tree	Memory Usage(per Machine)
1	627.8 s	176GB
2	311 s	87GB
4	156 s	43GB
8	80 s	22GB
16	42 s	11GB

From the results, we find that LightGBM performs linear speed up in parallel learning.

GPU Experiments

Refer to GPU Performance.

CHAPTER 6

Parameters

This is a page contains all parameters in LightGBM.

List of other Helpful Links

- *Python API*
- *Parameters Tuning*

External Links

- Laurae++ Interactive Documentation

Update of 08/04/2017

Default values for the following parameters have changed:

- min_data_in_leaf = 100 => 20
- min_sum_hessian_in_leaf = 10 => 1e-3
- num_leaves = 127 => 31
- num_iterations = 10 => 100

Parameter Format

The parameter format is `key1=value1 key2=value2 ...`. And parameters can be set both in config file and command line. By using command line, parameters should not have spaces before and after =. By using config files, one line can only contain one parameter. you can use # to comment. If one parameter appears in both command line and config file, LightGBM will use the parameter in command line.

Core Parameters

- `config`, default=`""`, type=string, alias=`config_file`

  - path of config file

- `task`, default=`train`, type=enum, options=`train`,`prediction`
  - `train` for training
  - `prediction` for prediction.
  - `convert_model` for converting model file into if-else format, see more information in *Convert model parameters*
- `application`, default=`regression`, type=enum, options=`regression`,`regression_l1`,`huber`,`fair`,`poisson`,`bin`
  alias=`objective`,`app`
  - `regression`, regression application
    * `regression_l2`, L2 loss, alias=`mean_squared_error`,`mse`
    * `regression_l1`, L1 loss, alias=`mean_absolute_error`,`mae`
    * `huber`, Huber loss
    * `fair`, Fair loss
    * `poisson`, Poisson regression
  - `binary`, binary classification application
  - `lambdarank`, lambdarank application
    * The label should be `int` type in lambdarank tasks, and larger number represent the higher relevance (e.g. 0:bad, 1:fair, 2:good, 3:perfect).
    * `label_gain` can be used to set the gain(weight) of `int` label.
  - `multiclass`, multi-class classification application, should set `num_class` as well
- `boosting`, default=`gbdt`, type=enum, options=`gbdt`,`rf`,`dart`,`goss`, alias=`boost`,`boosting_type`
  - `gbdt`, traditional Gradient Boosting Decision Tree
  - `rf`, Random Forest
  - `dart`, Dropouts meet Multiple Additive Regression Trees
  - `goss`, Gradient-based One-Side Sampling
- `data`, default=`""`, type=string, alias=`train`,`train_data`
  - training data, LightGBM will train from this data
- `valid`, default=`""`, type=multi-string, alias=`test`,`valid_data`,`test_data`
  - validation/test data, LightGBM will output metrics for these data
  - support multi validation data, separate by `,`
- `num_iterations`, default=`100`, type=int, alias=`num_iteration`,`num_tree`,`num_trees`,`num_round`,`num_rounds`
  - number of boosting iterations
  - note: For python/R package, **this parameter is ignored**, use `num_boost_round` (Python) or `nrounds` (R) input arguments of `train` and `cv` methods instead
  - note: internally, LightGBM constructs `num_class * num_iterations` trees for `multiclass` problems
- `learning_rate`, default=`0.1`, type=double, alias=`shrinkage_rate`
  - shrinkage rate

  - in `dart`, it also affects normalization weights of dropped trees
- `num_leaves`, default=`31`, type=int, alias=`num_leaf`
  - number of leaves in one tree
- `tree_learner`, default=`serial`, type=enum, options=`serial`,`feature`,`data`
  - `serial`, single machine tree learner
  - `feature`, feature parallel tree learner
  - `data`, data parallel tree learner
  - Refer to *Parallel Learning Guide* to get more details.
- `num_threads`, default=OpenMP_default, type=int, alias=`num_thread`,`nthread`
  - Number of threads for LightGBM.
  - For the best speed, set this to the number of **real CPU cores**, not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core).
  - Do not set it too large if your dataset is small (do not use 64 threads for a dataset with 10,000 for instance).
  - Be aware a task manager or any similar CPU monitoring tool might report cores not being fully utilized. This is normal.
  - For parallel learning, should not use full CPU cores since this will cause poor performance for the network.
- `device`, default=`cpu`, options=`cpu`,`gpu`
  - Choose device for the tree learning, can use gpu to achieve the faster learning.
  - Note: 1. Recommend use the smaller `max_bin`(e.g `63`) to get the better speed up. 2. For the faster speed, GPU use 32-bit float point to sum up by default, may affect the accuracy for some tasks. You can set `gpu_use_dp=true` to enable 64-bit float point, but it will slow down the training. 3. Refer to *Installation Guide* to build with GPU .

Learning Control Parameters

- `max_depth`, default=`-1`, type=int
  - Limit the max depth for tree model. This is used to deal with overfit when #data is small. Tree still grow by leaf-wise.
  - `< 0` means no limit
- `min_data_in_leaf`, default=`20`, type=int, alias=`min_data_per_leaf` , `min_data`
  - Minimal number of data in one leaf. Can use this to deal with over-fit.
- `min_sum_hessian_in_leaf`, default=`1e-3`, type=double, alias=`min_sum_hessian_per_leaf`, `min_sum_hessian`, `min_hessian`
  - Minimal sum hessian in one leaf. Like `min_data_in_leaf`, can use this to deal with over-fit.
- `feature_fraction`, default=`1.0`, type=double, `0.0 < feature_fraction < 1.0`, alias=`sub_feature`
  - LightGBM will random select part of features on each iteration if `feature_fraction` smaller than `1.0`. For example, if set to `0.8`, will select 80% features before training each tree.
  - Can use this to speed up training

  - Can use this to deal with over-fit

- `feature_fraction_seed`, default=`2`, type=int
  - Random seed for feature fraction.
- `bagging_fraction`, default=`1.0`, type=double, , `0.0 < bagging_fraction < 1.0`, alias=`sub_row`
  - Like `feature_fraction`, but this will random select part of data without resampling
  - Can use this to speed up training
  - Can use this to deal with over-fit
  - Note: To enable bagging, should set `bagging_freq` to a non zero value as well
- `bagging_freq`, default=`0`, type=int
  - Frequency for bagging, `0` means disable bagging. `k` means will perform bagging at every `k` iteration.
  - Note: To enable bagging, should set `bagging_fraction` as well
- `bagging_seed` , default=`3`, type=int
  - Random seed for bagging.
- `early_stopping_round` , default=`0`, type=int, alias=`early_stopping_rounds`,`early_stopping`
  - Will stop training if one metric of one validation data doesn't improve in last `early_stopping_round` rounds.
- `lambda_l1` , default=`0`, type=double
  - l1 regularization
- `lambda_l2` , default=`0`, type=double
  - l2 regularization
- `min_gain_to_split` , default=`0`, type=double
  - The minimal gain to perform split
- `drop_rate`, default=`0.1`, type=double
  - only used in `dart`
- `skip_drop`, default=`0.5`, type=double
  - only used in `dart`, probability of skipping drop
- `max_drop`, default=`50`, type=int
  - only used in `dart`, max number of dropped trees on one iteration. `<=0` means no limit.
- `uniform_drop`, default=`false`, type=bool
  - only used in `dart`, true if want to use uniform drop
- `xgboost_dart_mode`, default=`false`, type=bool
  - only used in `dart`, true if want to use xgboost dart mode
- `drop_seed`, default=`4`, type=int
  - only used in `dart`, used to random seed to choose dropping models.
- `top_rate`, default=`0.2`, type=double
  - only used in `goss`, the retain ratio of large gradient data

- `other_rate`, default=`0.1`, type=int
  - only used in `goss`, the retain ratio of small gradient data
- `max_cat_group`, default=`64`, type=int
  - use for the categorical features.
  - When #catogory is large, finding the split point on it is easily over-fitting. So LightGBM merges them into `max_cat_group` groups, and finds the split points on the group boundaries.
- `min_data_per_group`, default=`10`, type=int
  - Min number of data per categorical group.
- `max_cat_threshold`, default=`256`, type=int
  - use for the categorical features. Limit the max threshold points in categorical features.
- `min_cat_smooth`, default=`5`, type=double
  - use for the categorical features. Refer to the descrption in paramater `cat_smooth_ratio`.
- `max_cat_smooth`, default=`100`, type=double
  - use for the categorical features. Refer to the descrption in paramater `cat_smooth_ratio`.
- `cat_smooth_ratio`, default=`0.01`, type=double
  - use for the categorical features. This can reduce the effect of noises in categorical features, especially for categories with few data.
  - The smooth denominator is `a = min(max_cat_smooth, max(min_cat_smooth, num_data/num_category*cat_smooth_ratio))`.
  - The smooth numerator is `b = a * sum_gradient / sum_hessian`.

IO Parameters

- `max_bin`, default=`255`, type=int
  - max number of bin that feature values will bucket in. Small bin may reduce training accuracy but may increase general power (deal with over-fit).
  - LightGBM will auto compress memory according `max_bin`. For example, LightGBM will use `uint8_t` for feature value if `max_bin=255`.
- `min_data_in_bin`, default=`5`, type=int
  - min number of data inside one bin, use this to avoid one-data-one-bin (may over-fitting).
- `data_random_seed`, default=`1`, type=int
  - random seed for data partition in parallel learning(not include feature parallel).
- `output_model`, default=`LightGBM_model.txt`, type=string, alias=`model_output`,`model_out`
  - file name of output model in training.
- `input_model`, default=`""`, type=string, alias=`model_input`,`model_in`
  - file name of input model.
  - for prediction task, will prediction data using this model.
  - for train task, will continued train from this model.

- `output_result`, default=`LightGBM_predict_result.txt`, type=string, alias=`predict_result`,`prediction_result`
  - file name of prediction result in prediction task.
- `is_pre_partition`, default=`false`, type=bool
  - used for parallel learning(not include feature parallel).
  - `true` if training data are pre-partitioned, and different machines using different partition.
- `is_sparse`, default=`true`, type=bool, alias=`is_enable_sparse`
  - used to enable/disable sparse optimization. Set to `false` to disable sparse optimization.
- `two_round`, default=`false`, type=bool, alias=`two_round_loading`,`use_two_round_loading`
  - by default, LightGBM will map data file to memory and load features from memory. This will provide faster data loading speed. But it may out of memory when the data file is very big.
  - set this to `true` if data file is too big to fit in memory.
- `save_binary`, default=`false`, type=bool, alias=`is_save_binary`,`is_save_binary_file`
  - set this to `true` will save the data set(include validation data) to a binary file. Speed up the data loading speed for the next time.
- `verbosity`, default=`1`, type=int, alias=`verbose`
  - `<0` = Fatel, `=0` = Error(Warn), `>0` = Info
- `header`, default=`false`, type=bool, alias=`has_header`
  - `true` if input data has header
- `label`, default=`""`, type=string, alias=`label_column`
  - specific the label column
  - Use number for index, e.g. `label=0` means column_0 is the label
  - Add a prefix `name:` for column name, e.g. `label=name:is_click`
- `weight`, default=`""`, type=string, alias=`weight_column`
  - specific the weight column
  - Use number for index, e.g. `weight=0` means column_0 is the weight
  - Add a prefix `name:` for column name, e.g. `weight=name:weight`
  - Note: Index start from `0`. And it doesn't count the label column when passing type is Index. e.g. when label is column_0, and weight is column_1, the correct parameter is `weight=0`.
- `query`, default=`""`, type=string, alias=`query_column`,`group`,`group_column`
  - specific the query/group id column
  - Use number for index, e.g. `query=0` means column_0 is the query id
  - Add a prefix `name:` for column name, e.g. `query=name:query_id`
  - Note: Data should group by query_id. Index start from `0`. And it doesn't count the label column when passing type is Index. e.g. when label is column_0, and query_id is column_1, the correct parameter is `query=0`.
- `ignore_column`, default=`""`, type=string, alias=`ignore_feature`,`blacklist`
  - specific some ignore columns in training

  - Use number for index, e.g. `ignore_column=0,1,2` means column_0, column_1 and column_2 will be ignored.
  - Add a prefix `name:` for column name, e.g. `ignore_column=name:c1,c2,c3` means c1, c2 and c3 will be ignored.
  - Note: Index start from `0`. And it doesn't count the label column.
- `categorical_feature`, default=`""`, type=string, alias=`categorical_column`,`cat_feature`,`cat_column`
  - specific categorical features
  - Use number for index, e.g. `categorical_feature=0,1,2` means column_0, column_1 and column_2 are categorical features.
  - Add a prefix `name:` for column name, e.g. `categorical_feature=name:c1,c2,c3` means c1, c2 and c3 are categorical features.
  - Note: Only support categorical with `int` type (Note: the negative values will be treated as Missing values). Index start from `0`. And it doesn't count the label column.
- `predict_raw_score`, default=`false`, type=bool, alias=`raw_score`,`is_predict_raw_score`
  - only used in prediction task
  - Set to `true` will only predict the raw scores.
  - Set to `false` will transformed score
- `predict_leaf_index`, default=`false`, type=bool, alias=`leaf_index`,`is_predict_leaf_index`
  - only used in prediction task
  - Set to `true` to predict with leaf index of all trees
- `predict_contrib`, default=`false`, type=bool, alias=`contrib`,`is_predict_contrib`
  - only used in prediction task
  - Set to `true` to estimate SHAP values, which represent how each feature contributed to each prediction. Produces number of features + 1 values where the last value is the expected value of the model output over the training data.
- `bin_construct_sample_cnt`, default=`200000`, type=int
  - Number of data that sampled to construct histogram bins.
  - Will give better training result when set this larger. But will increase data loading time.
  - Set this to larger value if data is very sparse.
- `num_iteration_predict`, default=`-1`, type=int
  - only used in prediction task, used to how many trained iterations will be used in prediction.
  - `<= 0` means no limit
- `pred_early_stop`, default=`false`, type=bool
  - Set to `true` will use early-stopping to speed up the prediction. May affect the accuracy.
- `pred_early_stop_freq`, default=`10`, type=int
  - The frequency of checking early-stopping prediction.
- `pred_early_stop_margin`, default=`10.0`, type=double
  - The Threshold of margin in early-stopping prediction.

- `use_missing`, default=`true`, type=bool
  - Set to `false` will disable the special handle of missing value.
- `zero_as_missing`, default=`false`, type=bool
  - Set to `true` will treat all zero as missing values (including the unshown values in libsvm/sparse matrics).
  - Set to `false` will use `na` to represent missing values.
- `init_score_file`, default=`""`, type=string
  - Path of training initial score file, `""` will use `train_data_file+".init"` (if exists).
- `valid_init_score_file`, default=`""`, type=multi-string
  - Path of validation initial score file, `""` will use `valid_data_file+".init"` (if exists).
  - separate by `,` for multi-validation data

Objective Parameters

- `sigmoid`, default=`1.0`, type=double
  - parameter for sigmoid function. Will be used in binary classification and lambdarank.
- `huber_delta`, default=`1.0`, type=double
  - parameter for Huber loss. Will be used in regression task.
- `fair_c`, default=`1.0`, type=double
  - parameter for Fair loss. Will be used in regression task.
- `gaussian_eta`, default=`1.0`, type=double
  - parameter to control the width of Gaussian function. Will be used in l1 and huber regression loss.
- `poission_max_delta_step`, default=`0.7`, type=double
  - parameter used to safeguard optimization
- `scale_pos_weight`, default=`1.0`, type=double
  - weight of positive class in binary classification task
- `boost_from_average`, default=`true`, type=bool
  - adjust initial score to the mean of labels for faster convergence, only used in Regression task.
- `is_unbalance`, default=`false`, type=bool
  - used in binary classification. Set this to `true` if training data are unbalance.
- `max_position`, default=`20`, type=int
  - used in lambdarank, will optimize NDCG at this position.
- `label_gain`, default=`0,1,3,7,15,31,63,...`, type=multi-double
  - used in lambdarank, relevant gain for labels. For example, the gain of label `2` is `3` if using default label gains.
  - Separate by `,`
- `num_class`, default=`1`, type=int, alias=`num_classes`
  - only used in multi-class classification

Metric Parameters

- metric, default={l2 for regression}, {binary_logloss for binary classification},{ndcg for lambdarank}, type=multi-enum, options=l1,l2,ndcg,auc,binary_logloss,binary_error...
  - l1, absolute loss, alias=mean_absolute_error, mae
  - l2, square loss, alias=mean_squared_error, mse
  - l2_root, root square loss, alias=root_mean_squared_error, rmse
  - huber, Huber loss
  - fair, Fair loss
  - poisson, Poisson regression
  - ndcg, NDCG
  - map, MAP
  - auc, AUC
  - binary_logloss, log loss
  - binary_error. For one sample 0 for correct classification, 1 for error classification.
  - multi_logloss, log loss for mulit-class classification
  - multi_error. error rate for mulit-class classification
  - Support multi metrics, separate by ,
- metric_freq, default=1, type=int
  - frequency for metric output
- is_training_metric, default=false, type=bool
  - set this to true if need to output metric result of training
- ndcg_at, default=1,2,3,4,5, type=multi-int, alias=ndcg_eval_at,eval_at
  - NDCG evaluation position, separate by ,

Network Parameters

Following parameters are used for parallel learning, and only used for base(socket) version.

- num_machines, default=1, type=int, alias=num_machine
  - Used for parallel learning, the number of machines for parallel learning application
  - Need to set this in both socket and mpi version.
- local_listen_port, default=12400, type=int, alias=local_port
  - TCP listen port for local machines.
  - Should allow this port in firewall setting before training.
- time_out, default=120, type=int
  - Socket time-out in minutes.
- machine_list_file, default="", type=string

- File that list machines for this parallel learning application
- Each line contains one IP and one port for one machine. The format is `ip port`, separate by space.

GPU Parameters

- `gpu_platform_id`, default=`-1`, type=int
  - OpenCL platform ID. Usually each GPU vendor exposes one OpenCL platform.
  - Default value is -1, using the system-wide default platform.
- `gpu_device_id`, default=`-1`, type=int
  - OpenCL device ID in the specified platform. Each GPU in the selected platform has a unique device ID.
  - Default value is -1, using the default device in the selected platform.
- `gpu_use_dp`, default=`false`, type=bool
  - Set to true to use double precision math on GPU (default using single precision).

Convert Model Parameters

This feature is only supported in command line version yet.

- `convert_model_language`, default=`""`, type=string
  - only `cpp` is supported yet.
  - if `convert_model_language` is set when `task` is set to `train`, the model will also be converted.
- `convert_model`, default=`"gbdt_prediction.cpp"`, type=string
  - output file name of converted model.

Others

Continued Training with Input Score

LightGBM support continued train with initial score. It uses an additional file to store these initial score, like the following:

```
0.5
-0.1
0.9
...
```

It means the initial score of first data is `0.5`, second is `-0.1`, and so on. The initial score file corresponds with data file line by line, and has per score per line. And if the name of data file is “train.txt”, the initial score file should be named as “train.txt.init” and in the same folder as the data file. And LightGBM will auto load initial score file if it exists.

Weight Data

LightGBM support weighted training. It uses an additional file to store weight data, like the following:

```
1.0
0.5
0.8
...
```

It means the weight of first data is `1.0`, second is `0.5`, and so on. The weight file corresponds with data file line by line, and has per weight per line. And if the name of data file is "train.txt", the weight file should be named as "train.txt.weight" and in the same folder as the data file. And LightGBM will auto load weight file if it exists.

update: You can specific weight column in data file now. Please refer to parameter `weight` in above.

Query Data

For LambdaRank learning, it needs query information for training data. LightGBM use an additional file to store query data. Following is an example:

```
27
18
67
...
```

It means first `27` lines samples belong one query and next `18` lines belong to another, and so on.(**Note: data should order by query**) If name of data file is "train.txt", the query file should be named as "train.txt.query" and in same folder of training data. LightGBM will load the query file automatically if it exists.

You can specific query/group id in data file now. Please refer to parameter `group` in above.

CHAPTER 7

Parameters Tuning

This is a page contains all parameters in LightGBM.

List of other Helpful Links

- *Parameters*
- *Python API*

Tune Parameters for the Leaf-wise (Best-first) Tree

LightGBM uses the *leaf-wise* tree growth algorithm, while many other popular tools use depth-wise tree growth. Compared with depth-wise growth, the leaf-wise algorithm can convenge much faster. However, the leaf-wise growth may be over-fitting if not used with the appropriate parameters.

To get good results using a leaf-wise tree, these are some important parameters:

1. `num_leaves`. This is the main parameter to control the complexity of the tree model. Theoretically, we can set `num_leaves = 2^(max_depth)` to convert from depth-wise tree. However, this simple conversion is not good in practice. The reason is, when number of leaves are the same, the leaf-wise tree is much deeper than depth-wise tree. As a result, it may be over-fitting. Thus, when trying to tune the `num_leaves`, we should let it be smaller than `2^(max_depth)`. For example, when the `max_depth=6` the depth-wise tree can get good accuracy, but setting `num_leaves` to `127` may cause over-fitting, and setting it to `70` or `80` may get better accuracy than depth-wise. Actually, the concept `depth` can be forgotten in leaf-wise tree, since it doesn't have a correct mapping from `leaves` to `depth`.
2. `min_data_in_leaf`. This is a very important parameter to deal with over-fitting in leaf-wise tree. Its value depends on the number of training data and `num_leaves`. Setting it to a large value can avoid growing too deep a tree, but may cause under-fitting. In practice, setting it to hundreds or thousands is enough for a large dataset.
3. `max_depth`. You also can use `max_depth` to limit the tree depth explicitly.

For Faster Speed

- Use bagging by setting `bagging_fraction` and `bagging_freq`
- Use feature sub-sampling by setting `feature_fraction`
- Use small `max_bin`
- Use `save_binary` to speed up data loading in future learning
- Use parallel learning, refer to *Parallel Learning Guide*.

For Better Accuracy

- Use large `max_bin` (may be slower)
- Use small `learning_rate` with large `num_iterations`
- Use large `num_leaves`(may cause over-fitting)
- Use bigger training data
- Try `dart`

Deal with Over-fitting

- Use small `max_bin`
- Use small `num_leaves`
- Use `min_data_in_leaf` and `min_sum_hessian_in_leaf`
- Use bagging by set `bagging_fraction` and `bagging_freq`
- Use feature sub-sampling by set `feature_fraction`
- Use bigger training data
- Try `lambda_l1`, `lambda_l2` and `min_gain_to_split` to regularization
- Try `max_depth` to avoid growing deep tree

CHAPTER 8

Python API

Data Structure API

class `lightgbm.`**`Dataset`** (*data*, *label=None*, *max_bin=255*, *reference=None*, *weight=None*, *group=None*, *silent=False*, *feature_name='auto'*, *categorical_feature='auto'*, *params=None*, *free_raw_data=True*)

Bases: `object`

Dataset in LightGBM.

Constract Dataset.

Parameters

- **`data`** (*`string, numpy array or scipy.sparse`*) – Data source of Dataset. If string, it represents the path to txt file.
- **`label`** (*`list or numpy 1-D array, optional (default=None)`*) – Label of the data.
- **`max_bin`** (*`int, optional (default=255)`*) – Max number of discrete bins for features.
- **`reference`** (*`Dataset or None, optional (default=None)`*) – If this is Dataset for validation, training data should be used as reference.
- **`weight`** (*`list, numpy 1-D array or None, optional (default=None)`*) – Weight for each instance.
- **`group`** (*`list, numpy 1-D array or None, optional (default=None)`*) – Group/query size for Dataset.
- **`silent`** (*`bool, optional (default=False)`*) – Whether to print messages during construction.
- **`feature_name`** (*`list of strings or 'auto', optional (default="auto")`*) – Feature names. If 'auto' and data is pandas DataFrame, data columns names are used.

- **`categorical_feature`** (*`list of strings or int, or 'auto', optional (default="auto")`*) – Categorical features. If list of int, interpreted as indices. If list of strings, interpreted as feature names (need to specify `feature_name` as well). If 'auto' and data is pandas DataFrame, pandas categorical columns are used.
- **`params`** (*`dict or None, optional (default=None)`*) – Other parameters.
- **`free_raw_data`** (*`bool, optional (default=True)`*) – If True, raw data is freed after constructing inner Dataset.

`construct`()
Lazy init.

Returns **self** – Returns self.

Return type *Dataset*

`create_valid`(*data*, *label=None*, *weight=None*, *group=None*, *silent=False*, *params=None*)
Create validation data align with current Dataset.

Parameters

- **`data`** (*`string, numpy array or scipy.sparse`*) – Data source of Dataset. If string, it represents the path to txt file.
- **`label`** (*`list or numpy 1-D array, optional (default=None)`*) – Label of the training data.
- **`weight`** (*`list, numpy 1-D array or None, optional (default=None)`*) – Weight for each instance.
- **`group`** (*`list, numpy 1-D array or None, optional (default=None)`*) – Group/query size for Dataset.
- **`silent`** (*`bool, optional (default=False)`*) – Whether to print messages during construction.
- **`params`** (*`dict or None, optional (default=None)`*) – Other parameters.

Returns **self** – Returns self.

Return type *Dataset*

`get_field`(*field_name*)
Get property from the Dataset.

Parameters **`field_name`** (*`string`*) – The field name of the information.

Returns **info** – A numpy array with information from the Dataset.

Return type numpy array

`get_group`()
Get the group of the Dataset.

Returns **group** – Group size of each group.

Return type numpy array

`get_init_score`()
Get the initial score of the Dataset.

Returns **init_score** – Init score of Booster.

Return type numpy array

`get_label`()
Get the label of the Dataset.

Returns **label** – The label information from the Dataset.

Return type numpy array

`get_ref_chain`(*ref_limit=100*)
Get a chain of Dataset objects, starting with r, then going to r.reference if exists, then to r.reference.reference, etc. until we hit `ref_limit` or a reference loop.

Parameters **`ref_limit`** (*`int, optional (default=100)`*) – The limit number of references.

Returns **ref_chain** – Chain of references of the Datasets.

Return type set of Dataset

`get_weight`()
Get the weight of the Dataset.

Returns **weight** – Weight for each data point from the Dataset.

Return type numpy array

`num_data`()
Get the number of rows in the Dataset.

Returns **number_of_rows** – The number of rows in the Dataset.

Return type int

`num_feature`()
Get the number of columns (features) in the Dataset.

Returns **number_of_columns** – The number of columns (features) in the Dataset.

Return type int

`save_binary`(*filename*)
Save Dataset to binary file.

Parameters **`filename`** (*`string`*) – Name of the output file.

`set_categorical_feature`(*categorical_feature*)
Set categorical features.

Parameters **`categorical_feature`** (*`list of int or strings`*) – Names or indices of categorical features.

`set_feature_name`(*feature_name*)
Set feature name.

Parameters **`feature_name`** (*`list of strings`*) – Feature names.

`set_field`(*field_name*, *data*)
Set property into the Dataset.

Parameters

- **`field_name`** (*`string`*) – The field name of the information.
- **`data`** (*`list, numpy array or None`*) – The array of data to be set.

`set_group`(*group*)
Set group size of Dataset (used for ranking).

Parameters **group** (*list, numpy array or None*) – Group size of each group.

set_init_score (*init_score*)
Set init score of Booster to start from.

Parameters **init_score** (*list, numpy array or None*) – Init score for Booster.

set_label (*label*)
Set label of Dataset

Parameters **label** (*list, numpy array or None*) – The label information to be set into Dataset.

set_reference (*reference*)
Set reference Dataset.

Parameters **reference** (Dataset) – Reference that is used as a template to consturct the current Dataset.

set_weight (*weight*)
Set weight of each instance.

Parameters **weight** (*list, numpy array or None*) – Weight to be set for each data point.

subset (*used_indices*, *params=None*)
Get subset of current Dataset.

Parameters

- **used_indices** (*list of int*) – Indices used to create the subset.
- **params** (*dict or None, optional (default=None)*) – Other parameters.

Returns **subset** – Subset of the current Dataset.

Return type *Dataset*

class lightgbm.**Booster** (*params=None*, *train_set=None*, *model_file=None*, *silent=False*)
Bases: object

Booster in LightGBM.

Initialize the Booster.

Parameters

- **params** (*dict or None, optional (default=None)*) – Parameters for Booster.
- **train_set** (Dataset *or None, optional (default=None)*) – Training dataset.
- **model_file** (*string or None, optional (default=None)*) – Path to the model file.
- **silent** (*bool, optional (default=False)*) – Whether to print messages during construction.

add_valid (*data*, *name*)
Add validation data.

Parameters

- **data** (Dataset) – Validation data.
- **name** (*string*) – Name of validation data.

`attr`(*key*)

Get attribute string from the Booster.

Parameters **`key`** (*`string`*) – The name of the attribute.

Returns **value** – The attribute value. Returns None if attribute do not exist.

Return type string or None

`current_iteration`()

Get the index of the current iteration.

Returns **cur_iter** – The index of the current iteration.

Return type int

`dump_model`(*num_iteration=-1*)

Dump Booster to json format.

Parameters **`num_iteration`** (*`int, optional (default=-1)`*) – Index of the iteration that should to dumped. If <0, the best iteration (if exists) is dumped.

Returns **json_repr** – Json format of Booster.

Return type dict

`eval`(*data*, *name*, *feval=None*)

Evaluate for data.

Parameters

- **`data`** (`Dataset`) – Data for the evaluating.
- **`name`** (*`string`*) – Name of the data.
- **`feval`** (*`callable or None, optional (default=None)`*) – Custom evaluation function.

Returns **result** – List with evaluation results.

Return type list

`eval_train`(*feval=None*)

Evaluate for training data.

Parameters **`feval`** (*`callable or None, optional (default=None)`*) – Custom evaluation function.

Returns **result** – List with evaluation results.

Return type list

`eval_valid`(*feval=None*)

Evaluate for validation data.

Parameters **`feval`** (*`callable or None, optional (default=None)`*) – Custom evaluation function.

Returns **result** – List with evaluation results.

Return type list

`feature_importance`(*importance_type='split'*, *iteration=-1*)

Get feature importances.

Parameters **`importance_type`** (*`string, optional (default="split")`*) – How the importance is calculated. If "split", result contains numbers of times the feature is used in a model. If "gain", result contains total gains of splits which use the feature.

Returns **result** – Array with feature importances.

Return type numpy array

`feature_name`()
Get names of features.

Returns **result** – List with names of features.

Return type list

`free_dataset`()
Free Booster's Datasets.

`get_leaf_output`(*tree_id*, *leaf_id*)
Get the output of a leaf.

Parameters

- **`tree_id`** (*`int`*) – The index of the tree.
- **`leaf_id`** (*`int`*) – The index of the leaf in the tree.

Returns **result** – The output of the leaf.

Return type float

`num_feature`()
Get number of features.

Returns **num_feature** – The number of features.

Return type int

`params_str` **= None**
construct booster object

`predict`(*data*, *num_iteration=-1*, *raw_score=False*, *pred_leaf=False*, *pred_contrib=False*, *data_has_header=False*, *is_reshape=True*, *pred_parameter=None*)
Make a prediction.

Parameters

- **`data`** (*`string, numpy array or scipy.sparse`*) – Data source for prediction. If string, it represents the path to txt file.
- **`num_iteration`** (*`int, optional (default=-1)`*) – Iteration used for prediction. If <0, the best iteration (if exists) is used for prediction.
- **`raw_score`** (*`bool, optional (default=False)`*) – Whether to predict raw scores.
- **`pred_leaf`** (*`bool, optional (default=False)`*) – Whether to predict leaf index.
- **`pred_contrib`** (*`bool, optional (default=False)`*) – Whether to predict feature contributions.
- **`data_has_header`** (*`bool, optional (default=False)`*) – Whether the data has header. Used only if data is string.
- **`is_reshape`** (*`bool, optional (default=True)`*) – If True, result is reshaped to [nrow, ncol].

- **`pred_parameter`** (*`dict or None, optional (default=None)`*) – Other parameters for the prediction.

Returns **result** – Prediction result.

Return type numpy array

`reset_parameter`(*params*)
Reset parameters of Booster.

Parameters **`params`** (*`dict`*) – New parameters for Booster.

`rollback_one_iter`()
Rollback one iteration.

`save_model`(*filename*, *num_iteration=-1*)
Save Booster to file.

Parameters

- **`filename`** (*`string`*) – Filename to save Booster.
- **`num_iteration`** (*`int, optional (default=-1)`*) – Index of the iteration that should to saved. If <0, the best iteration (if exists) is saved.

`set_attr`(***kwargs*)
Set the attribute of the Booster.

Parameters **`**kwargs`** – The attributes to set. Setting a value to None deletes an attribute.

`set_train_data_name`(*name*)
Set the name to the training Dataset.

Parameters **`name`** (*`string`*) – Name for training Dataset.

`update`(*train_set=None*, *fobj=None*)
Update for one iteration.

Parameters

- **`train_set`** (*`Dataset or None, optional (default=None)`*) – Training data. If None, last training data is used.
- **`fobj`** (*`callable or None, optional (default=None)`*) – Customized objective function.

  For multi-class task, the score is group by class_id first, then group by row_id. If you want to get i-th row score in j-th class, the access way is score[j * num_data + i] and you should group grad and hess in this way as well.

Returns **is_finished** – Whether the update was successfully finished.

Return type bool

Training API

`lightgbm.`**`train`**(*params*, *train_set*, *num_boost_round=100*, *valid_sets=None*, *valid_names=None*, *fobj=None*, *feval=None*, *init_model=None*, *feature_name='auto'*, *categorical_feature='auto'*, *early_stopping_rounds=None*, *evals_result=None*, *verbose_eval=True*, *learning_rates=None*, *keep_training_booster=False*, *callbacks=None*)
Perform the training with given parameters.

Parameters

- **params** (*dict*) – Parameters for training.
- **train_set** (`Dataset`) – Data to be trained.
- **num_boost_round** (*int, optional (default=100)*) – Number of boosting iterations.
- **valid_sets** (*list of Datasets or None, optional (default=None)*) – List of data to be evaluated during training.
- **valid_names** (*list of string or None, optional (default=None)*) – Names of `valid_sets`.
- **fobj** (*callable or None, optional (default=None)*) – Customized objective function.
- **feval** (*callable or None, optional (default=None)*) – Customized evaluation function. Note: should return (eval_name, eval_result, is_higher_better) or list of such tuples.
- **init_model** (*string or None, optional (default=None)*) – Filename of LightGBM model or Booster instance used for continue training.
- **feature_name** (*list of strings or 'auto', optional (default="auto")*) – Feature names. If 'auto' and data is pandas DataFrame, data columns names are used.
- **categorical_feature** (*list of strings or int, or 'auto', optional (default="auto")*) – Categorical features. If list of int, interpreted as indices. If list of strings, interpreted as feature names (need to specify `feature_name` as well). If 'auto' and data is pandas DataFrame, pandas categorical columns are used.
- **early_stopping_rounds** (*int or None, optional (default=None)*) – Activates early stopping. The model will train until the validation score stops improving. Requires at least one validation data and one metric. If there's more than one, will check all of them. If early stopping occurs, the model will add `best_iteration` field.
- **evals_result** (*dict or None, optional (default=None)*) – This dictionary used to store all evaluation results of all the items in `valid_sets`.

  ### Example

  With a `valid_sets` = [valid_set, train_set], `valid_names` = ['eval', 'train'] and a `params` = ('metric':'logloss') returns: {'train': {'logloss': ['0.48253', '0.35953', ...]}, 'eval': {'logloss': ['0.480385', '0.357756', ...]}}.

- **verbose_eval** (*bool or int, optional (default=True)*) – Requires at least one validation data. If True, the eval metric on the valid set is printed at each boosting stage. If int, the eval metric on the valid set is printed at every `verbose_eval` boosting stage. The last boosting stage or the boosting stage found by using `early_stopping_rounds` is also printed.

  ### Example

  With `verbose_eval` = 4 and at least one item in evals, an evaluation metric is printed every 4 (instead of 1) boosting stages.

- **learning_rates** (*list, callable or None, optional (default=None)*) – List of learning rates for each boosting round or a customized function that calculates `learning_rate` in terms of current number of round (e.g. yields learning rate decay).
- **keep_training_booster** (*bool, optional (default=False)*) – Whether the returned Booster will be used to keep training. If False, the returned value will be converted into _InnerPredictor before returning. You can still use _InnerPredictor as `init_model` for future continue training.
- **callbacks** (*list of callables or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **booster** – The trained Booster model.

Return type *Booster*

`lightgbm.`**`cv`**(*params*, *train_set*, *num_boost_round=10*, *folds=None*, *nfold=5*, *stratified=True*, *shuffle=True*, *metrics=None*, *fobj=None*, *feval=None*, *init_model=None*, *feature_name='auto'*, *categorical_feature='auto'*, *early_stopping_rounds=None*, *fpreproc=None*, *verbose_eval=None*, *show_stdv=True*, *seed=0*, *callbacks=None*)

Perform the cross-validation with given paramaters.

Parameters

- **params** (*dict*) – Parameters for Booster.
- **train_set** (`Dataset`) – Data to be trained on.
- **num_boost_round** (*int, optional (default=10)*) – Number of boosting iterations.
- **folds** (*a generator or iterator of (train_idx, test_idx) tuples or None, optional (default=None)*) – The train and test indices for the each fold. This argument has highest priority over other data split arguments.
- **nfold** (*int, optional (default=5)*) – Number of folds in CV.
- **stratified** (*bool, optional (default=True)*) – Whether to perform stratified sampling.
- **shuffle** (*bool, optional (default=True)*) – Whether to shuffle before splitting data.
- **metrics** (*string, list of strings or None, optional (default=None)*) – Evaluation metrics to be monitored while CV. If not None, the metric in `params` will be overridden.
- **fobj** (*callable or None, optional (default=None)*) – Custom objective function.
- **feval** (*callable or None, optional (default=None)*) – Custom evaluation function.
- **init_model** (*string or None, optional (default=None)*) – Filename of LightGBM model or Booster instance used for continue training.
- **feature_name** (*list of strings or 'auto', optional (default="auto")*) – Feature names. If 'auto' and data is pandas DataFrame, data columns names are used.
- **categorical_feature** (*list of strings or int, or 'auto', optional (default="auto")*) – Categorical features. If list of int, interpreted as

indices. If list of strings, interpreted as feature names (need to specify `feature_name` as well). If 'auto' and data is pandas DataFrame, pandas categorical columns are used.

- **`early_stopping_rounds`** (*`int or None, optional (default=None)`*) – Activates early stopping. CV error needs to decrease at least every `early_stopping_rounds` round(s) to continue. Last entry in evaluation history is the one from best iteration.
- **`fpreproc`** (*`callable or None, optional (default=None)`*) – Preprocessing function that takes (dtrain, dtest, params) and returns transformed versions of those.
- **`verbose_eval`** (*`bool, int, or None, optional (default=None)`*) – Whether to display the progress. If None, progress will be displayed when np.ndarray is returned. If True, progress will be displayed at every boosting stage. If int, progress will be displayed at every given `verbose_eval` boosting stage.
- **`show_stdv`** (*`bool, optional (default=True)`*) – Whether to display the standard deviation in progress. Results are not affected by this parameter, and always contains std.
- **`seed`** (*`int, optional (default=0)`*) – Seed used to generate the folds (passed to numpy.random.seed).
- **`callbacks`** (*`list of callables or None, optional (default=None)`*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **eval_hist** – Evaluation history. The dictionary has the following format: {'metric1-mean': [values], 'metric1-stdv': [values], 'metric2-mean': [values], 'metric1-stdv': [values], ...}.

Return type dict

Scikit-learn API

class `lightgbm.`**`LGBMModel`**(*boosting_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=10, max_bin=255, subsample_for_bin=50000, objective=None, min_split_gain=0.0, min_child_weight=5, min_child_samples=10, subsample=1.0, subsample_freq=1, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=0, n_jobs=-1, silent=True, **kwargs*)

Bases: `object`

Implementation of the scikit-learn API for LightGBM.

Construct a gradient boosting model.

Parameters

- **`boosting_type`** (*`string, optional (default="gbdt")`*) – 'gbdt', traditional Gradient Boosting Decision Tree. 'dart', Dropouts meet Multiple Additive Regression Trees. 'goss', Gradient-based One-Side Sampling. 'rf', Random Forest.
- **`num_leaves`** (*`int, optional (default=31)`*) – Maximum tree leaves for base learners.
- **`max_depth`** (*`int, optional (default=-1)`*) – Maximum tree depth for base learners, -1 means no limit.
- **`learning_rate`** (*`float, optional (default=0.1)`*) – Boosting learning rate.

- **n_estimators** (*int, optional (default=10)*) – Number of boosted trees to fit.
- **max_bin** (*int, optional (default=255)*) – Number of bucketed bin for feature values.
- **subsample_for_bin** (*int, optional (default=50000)*) – Number of samples for constructing bins.
- **objective** (*string, callable or None, optional (default=None)*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). default: 'binary' for LGBMClassifier, 'lambdarank' for LGBMRanker.
- **min_split_gain** (*float, optional (default=0.)*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*int, optional (default=5)*) – Minimum sum of instance weight(hessian) needed in a child(leaf).
- **min_child_samples** (*int, optional (default=10)*) – Minimum number of data need in a child(leaf).
- **subsample** (*float, optional (default=1.)*) – Subsample ratio of the training instance.
- **subsample_freq** (*int, optional (default=1)*) – Frequence of subsample, <=0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, optional (default=0)*) – Random number seed.
- **n_jobs** (*int, optional (default=-1)*) – Number of parallel threads.
- **silent** (*bool, optional (default=True)*) – Whether to print messages while running boosting.
- ****kwargs** (*other parameters*) – Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

Note: **kwargs is not supported in sklearn, it may cause unexpected issues.

n_features_
int – The number of features of fitted model.

classes_
array of shape = [n_classes] – The class label array (only for classification problem).

n_classes_
int – The number of classes (only for classification problem).

best_score_
dict or None – The best score of fitted model.

best_iteration_
int or None – The best iteration of fitted model if `early_stopping_rounds` has been specified.

objective_
string or callable – The concrete objective used while fitting this model.

booster_
Booster – The underlying Booster of this model.

evals_result_
dict or None – The evaluation results if `early_stopping_rounds` has been specified.

feature_importances_
array of shape = [n_features] – The feature importances (the higher, the more important the feature).

Note: A custom objective function can be provided for the `objective` parameter. In this case, it should have the signature `objective(y_true, y_pred) -> grad, hess` or `objective(y_true, y_pred, group) -> grad, hess`:

y_true: array-like of shape = [n_samples] The target values.

y_pred: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The predicted values.

group: array-like Group/query data, used for ranking task.

grad: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The value of the gradient for each sample point.

hess: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The value of the second derivative for each sample point.

For multi-class task, the y_pred is group by class_id first, then group by row_id. If you want to get i-th row y_pred in j-th class, the access way is y_pred[j * num_data + i] and you should group grad and hess in this way as well.

---

apply(*X*, *num_iteration=0*)
Return the predicted leaf every tree for each sample.

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – Input features matrix.
- **num_iteration** (*int, optional (default=0)*) – Limit number of iterations in the prediction; defaults to 0 (use all trees).

Returns **X_leaves** – The predicted leaf every tree for each sample.

Return type array-like of shape = [n_samples, n_trees]

best_iteration_
Get the best iteration of fitted model.

best_score_
Get the best score of fitted model.

booster_
Get the underlying lightgbm Booster of this model.

evals_result_
Get the evaluation results.

feature_importances_

Get feature importances.

Note: Feature importance in sklearn interface used to normalize to 1, it's deprecated after 2.0.4 and same as Booster.feature_importance() now.

fit(*X*, *y*, *sample_weight=None*, *init_score=None*, *group=None*, *eval_set=None*, *eval_names=None*, *eval_sample_weight=None*, *eval_init_score=None*, *eval_group=None*, *eval_metric=None*, *early_stopping_rounds=None*, *verbose=True*, *feature_name='auto'*, *categorical_feature='auto'*, *callbacks=None*)

Build a gradient boosting model from the training set (X, y).

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – Input feature matrix.
- **y** (*array-like of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Weights of training data.
- **init_score** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Init score of training data.
- **group** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Group data of training data.
- **eval_set** (*list or None, optional (default=None)*) – A list of (X, y) tuple pairs to use as a validation sets for early-stopping.
- **eval_names** (*list of strings or None, optional (default=None)*) – Names of eval_set.
- **eval_sample_weight** (*list of arrays or None, optional (default=None)*) – Weights of eval data.
- **eval_init_score** (*list of arrays or None, optional (default=None)*) – Init score of eval data.
- **eval_group** (*list of arrays or None, optional (default=None)*) – Group data of eval data.
- **eval_metric** (*string, list of strings, callable or None, optional (default=None)*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note for more details.
- **early_stopping_rounds** (*int or None, optional (default=None)*) – Activates early stopping. The model will train until the validation score stops improving. Validation error needs to decrease at least every `early_stopping_rounds` round(s) to continue training.
- **verbose** (*bool, optional (default=True)*) – If True and an evaluation set is used, writes the evaluation progress.
- **feature_name** (*list of strings or 'auto', optional (default="auto")*) – Feature names. If 'auto' and data is pandas DataFrame, data columns names are used.

- **categorical_feature** (*list of strings or int, or 'auto', optional (default="auto")*) – Categorical features. If list of int, interpreted as indices. If list of strings, interpreted as feature names (need to specify `feature_name` as well). If 'auto' and data is pandas DataFrame, pandas categorical columns are used.
- **callbacks** (*list of callback functions or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **self** – Returns self.

Return type object

Note: Custom eval function expects a callable with following functions: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)`. Returns (eval_name, eval_result, is_bigger_better) or list of (eval_name, eval_result, is_bigger_better)

y_true: array-like of shape = [n_samples] The target values.

y_pred: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class) The predicted values.

weight: array-like of shape = [n_samples] The weight of samples.

group: array-like Group/query data, used for ranking task.

eval_name: str The name of evaluation.

eval_result: float The eval result.

is_bigger_better: bool Is eval result bigger better, e.g. AUC is bigger_better.

For multi-class task, the y_pred is group by class_id first, then group by row_id. If you want to get i-th row y_pred in j-th class, the access way is y_pred[j * num_data + i].

n_features_
Get the number of features of fitted model.

objective_
Get the concrete objective used while fitting this model.

predict (*X*, *raw_score=False*, *num_iteration=0*)
Return the predicted value for each sample.

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – Input features matrix.
- **raw_score** (*bool, optional (default=False)*) – Whether to predict raw scores.
- **num_iteration** (*int, optional (default=0)*) – Limit number of iterations in the prediction; defaults to 0 (use all trees).

Returns **predicted_result** – The predicted values.

Return type array-like of shape = [n_samples] or shape = [n_samples, n_classes]

class `lightgbm.`**`LGBMClassifier`**(*boosting_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=10, max_bin=255, subsample_for_bin=50000, objective=None, min_split_gain=0.0, min_child_weight=5, min_child_samples=10, subsample=1.0, subsample_freq=1, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=0, n_jobs=-1, silent=True, **kwargs*)

Bases: `lightgbm.sklearn.LGBMModel`, `object`

LightGBM classifier.

Construct a gradient boosting model.

Parameters

- **`boosting_type`** (*`string, optional (default="gbdt")`*) – 'gbdt', traditional Gradient Boosting Decision Tree. 'dart', Dropouts meet Multiple Additive Regression Trees. 'goss', Gradient-based One-Side Sampling. 'rf', Random Forest.
- **`num_leaves`** (*`int, optional (default=31)`*) – Maximum tree leaves for base learners.
- **`max_depth`** (*`int, optional (default=-1)`*) – Maximum tree depth for base learners, -1 means no limit.
- **`learning_rate`** (*`float, optional (default=0.1)`*) – Boosting learning rate.
- **`n_estimators`** (*`int, optional (default=10)`*) – Number of boosted trees to fit.
- **`max_bin`** (*`int, optional (default=255)`*) – Number of bucketed bin for feature values.
- **`subsample_for_bin`** (*`int, optional (default=50000)`*) – Number of samples for constructing bins.
- **`objective`** (*`string, callable or None, optional (default=None)`*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). default: 'binary' for LGBMClassifier, 'lambdarank' for LGBMRanker.
- **`min_split_gain`** (*`float, optional (default=0.)`*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **`min_child_weight`** (*`int, optional (default=5)`*) – Minimum sum of instance weight(hessian) needed in a child(leaf).
- **`min_child_samples`** (*`int, optional (default=10)`*) – Minimum number of data need in a child(leaf).
- **`subsample`** (*`float, optional (default=1.)`*) – Subsample ratio of the training instance.
- **`subsample_freq`** (*`int, optional (default=1)`*) – Frequence of subsample, <=0 means no enable.
- **`colsample_bytree`** (*`float, optional (default=1.)`*) – Subsample ratio of columns when constructing each tree.
- **`reg_alpha`** (*`float, optional (default=0.)`*) – L1 regularization term on weights.
- **`reg_lambda`** (*`float, optional (default=0.)`*) – L2 regularization term on weights.

- **random_state** (*int, optional (default=0)*) – Random number seed.
- **n_jobs** (*int, optional (default=-1)*) – Number of parallel threads.
- **silent** (*bool, optional (default=True)*) – Whether to print messages while running boosting.
- ****kwargs** (*other parameters*) – Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

Note: **kwargs is not supported in sklearn, it may cause unexpected issues.

n_features_
int – The number of features of fitted model.

classes_
array of shape = [n_classes] – The class label array (only for classification problem).

n_classes_
int – The number of classes (only for classification problem).

best_score_
dict or None – The best score of fitted model.

best_iteration_
int or None – The best iteration of fitted model if `early_stopping_rounds` has been specified.

objective_
string or callable – The concrete objective used while fitting this model.

booster_
Booster – The underlying Booster of this model.

evals_result_
dict or None – The evaluation results if `early_stopping_rounds` has been specified.

feature_importances_
array of shape = [n_features] – The feature importances (the higher, the more important the feature).

Note: A custom objective function can be provided for the `objective` parameter. In this case, it should have the signature `objective(y_true, y_pred) -> grad, hess` or `objective(y_true, y_pred, group) -> grad, hess`:

y_true: array-like of shape = [n_samples] The target values.

y_pred: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The predicted values.

group: array-like Group/query data, used for ranking task.

grad: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The value of the gradient for each sample point.

hess: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The value of the second derivative for each sample point.

For multi-class task, the y_pred is group by class_id first, then group by row_id. If you want to get i-th row y_pred in j-th class, the access way is y_pred[j * num_data + i] and you should group grad and hess in this way as well.

`classes_`
Get the class label array.

`fit`(*X*, *y*, *sample_weight=None*, *init_score=None*, *eval_set=None*, *eval_names=None*, *eval_sample_weight=None*, *eval_init_score=None*, *eval_metric='logloss'*, *early_stopping_rounds=None*, *verbose=True*, *feature_name='auto'*, *categorical_feature='auto'*, *callbacks=None*)
Build a gradient boosting model from the training set (X, y).

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – Input feature matrix.
- **y** (*array-like of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Weights of training data.
- **init_score** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Init score of training data.
- **group** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Group data of training data.
- **eval_set** (*list or None, optional (default=None)*) – A list of (X, y) tuple pairs to use as a validation sets for early-stopping.
- **eval_names** (*list of strings or None, optional (default=None)*) – Names of eval_set.
- **eval_sample_weight** (*list of arrays or None, optional (default=None)*) – Weights of eval data.
- **eval_init_score** (*list of arrays or None, optional (default=None)*) – Init score of eval data.
- **eval_group** (*list of arrays or None, optional (default=None)*) – Group data of eval data.
- **eval_metric** (*string, list of strings, callable or None, optional (default="logloss")*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note for more details.
- **early_stopping_rounds** (*int or None, optional (default=None)*) – Activates early stopping. The model will train until the validation score stops improving. Validation error needs to decrease at least every `early_stopping_rounds` round(s) to continue training.
- **verbose** (*bool, optional (default=True)*) – If True and an evaluation set is used, writes the evaluation progress.
- **feature_name** (*list of strings or 'auto', optional (default="auto")*) – Feature names. If 'auto' and data is pandas DataFrame, data columns names are used.
- **categorical_feature** (*list of strings or int, or 'auto', optional (default="auto")*) – Categorical features. If list of int, interpreted as indices. If list of strings, interpreted as feature names (need to specify `feature_name` as well). If 'auto' and data is pandas DataFrame, pandas categorical columns are used.

- **callbacks** (*list of callback functions or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **self** – Returns self.

Return type object

Note: Custom eval function expects a callable with following functions: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)`. Returns (eval_name, eval_result, is_bigger_better) or list of (eval_name, eval_result, is_bigger_better)

y_true: array-like of shape = [n_samples] The target values.

y_pred: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class) The predicted values.

weight: array-like of shape = [n_samples] The weight of samples.

group: array-like Group/query data, used for ranking task.

eval_name: str The name of evaluation.

eval_result: float The eval result.

is_bigger_better: bool Is eval result bigger better, e.g. AUC is bigger_better.

For multi-class task, the y_pred is group by class_id first, then group by row_id. If you want to get i-th row y_pred in j-th class, the access way is y_pred[j * num_data + i].

`n_classes_`
Get the number of classes.

`predict_proba` (*X*, *raw_score=False*, *num_iteration=0*)
Return the predicted probability for each class for each sample.

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – Input features matrix.
- **raw_score** (*bool, optional (default=False)*) – Whether to predict raw scores.
- **num_iteration** (*int, optional (default=0)*) – Limit number of iterations in the prediction; defaults to 0 (use all trees).

Returns **predicted_probability** – The predicted probability for each class for each sample.

Return type array-like of shape = [n_samples, n_classes]

class `lightgbm.`**`LGBMRegressor`**(*boosting_type='gbdt'*, *num_leaves=31*, *max_depth=-1*, *learning_rate=0.1*, *n_estimators=10*, *max_bin=255*, *subsample_for_bin=50000*, *objective=None*, *min_split_gain=0.0*, *min_child_weight=5*, *min_child_samples=10*, *subsample=1.0*, *subsample_freq=1*, *colsample_bytree=1.0*, *reg_alpha=0.0*, *reg_lambda=0.0*, *random_state=0*, *n_jobs=-1*, *silent=True*, ***kwargs*)

Bases: `lightgbm.sklearn.LGBMModel`, `object`

LightGBM regressor.

Construct a gradient boosting model.

Parameters

- **boosting_type** (*string, optional (default="gbdt")*) – 'gbdt', traditional Gradient Boosting Decision Tree. 'dart', Dropouts meet Multiple Additive Regression Trees. 'goss', Gradient-based One-Side Sampling. 'rf', Random Forest.
- **num_leaves** (*int, optional (default=31)*) – Maximum tree leaves for base learners.
- **max_depth** (*int, optional (default=-1)*) – Maximum tree depth for base learners, -1 means no limit.
- **learning_rate** (*float, optional (default=0.1)*) – Boosting learning rate.
- **n_estimators** (*int, optional (default=10)*) – Number of boosted trees to fit.
- **max_bin** (*int, optional (default=255)*) – Number of bucketed bin for feature values.
- **subsample_for_bin** (*int, optional (default=50000)*) – Number of samples for constructing bins.
- **objective** (*string, callable or None, optional (default=None)*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). default: 'binary' for LGBMClassifier, 'lambdarank' for LGBMRanker.
- **min_split_gain** (*float, optional (default=0.)*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*int, optional (default=5)*) – Minimum sum of instance weight(hessian) needed in a child(leaf).
- **min_child_samples** (*int, optional (default=10)*) – Minimum number of data need in a child(leaf).
- **subsample** (*float, optional (default=1.)*) – Subsample ratio of the training instance.
- **subsample_freq** (*int, optional (default=1)*) – Frequence of subsample, <=0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, optional (default=0)*) – Random number seed.
- **n_jobs** (*int, optional (default=-1)*) – Number of parallel threads.
- **silent** (*bool, optional (default=True)*) – Whether to print messages while running boosting.
- ****kwargs** (*other parameters*) – Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

Note: **kwargs is not supported in sklearn, it may cause unexpected issues.

`n_features_`
int – The number of features of fitted model.

`classes_`
array of shape = [n_classes] – The class label array (only for classification problem).

`n_classes_`
int – The number of classes (only for classification problem).

`best_score_`
dict or None – The best score of fitted model.

`best_iteration_`
int or None – The best iteration of fitted model if `early_stopping_rounds` has been specified.

`objective_`
string or callable – The concrete objective used while fitting this model.

`booster_`
Booster – The underlying Booster of this model.

`evals_result_`
dict or None – The evaluation results if `early_stopping_rounds` has been specified.

`feature_importances_`
array of shape = [n_features] – The feature importances (the higher, the more important the feature).

Note: A custom objective function can be provided for the `objective` parameter. In this case, it should have the signature `objective(y_true, y_pred) -> grad, hess` or `objective(y_true, y_pred, group) -> grad, hess`:

y_true: array-like of shape = [n_samples] The target values.

y_pred: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The predicted values.

group: array-like Group/query data, used for ranking task.

grad: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The value of the gradient for each sample point.

hess: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The value of the second derivative for each sample point.

For multi-class task, the y_pred is group by class_id first, then group by row_id. If you want to get i-th row y_pred in j-th class, the access way is y_pred[j * num_data + i] and you should group grad and hess in this way as well.

`fit`(*X, y, sample_weight=None, init_score=None, eval_set=None, eval_names=None, eval_sample_weight=None, eval_init_score=None, eval_metric='l2', early_stopping_rounds=None, verbose=True, feature_name='auto', categorical_feature='auto', callbacks=None*)
Build a gradient boosting model from the training set (X, y).

Parameters

- **X** (*`array-like or sparse matrix of shape = [n_samples, n_features]`*) – Input feature matrix.

- **y** (*array-like of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Weights of training data.
- **init_score** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Init score of training data.
- **group** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Group data of training data.
- **eval_set** (*list or None, optional (default=None)*) – A list of (X, y) tuple pairs to use as a validation sets for early-stopping.
- **eval_names** (*list of strings or None, optional (default=None)*) – Names of eval_set.
- **eval_sample_weight** (*list of arrays or None, optional (default=None)*) – Weights of eval data.
- **eval_init_score** (*list of arrays or None, optional (default=None)*) – Init score of eval data.
- **eval_group** (*list of arrays or None, optional (default=None)*) – Group data of eval data.
- **eval_metric** (*string, list of strings, callable or None, optional (default="l2")*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note for more details.
- **early_stopping_rounds** (*int or None, optional (default=None)*) – Activates early stopping. The model will train until the validation score stops improving. Validation error needs to decrease at least every `early_stopping_rounds` round(s) to continue training.
- **verbose** (*bool, optional (default=True)*) – If True and an evaluation set is used, writes the evaluation progress.
- **feature_name** (*list of strings or 'auto', optional (default="auto")*) – Feature names. If 'auto' and data is pandas DataFrame, data columns names are used.
- **categorical_feature** (*list of strings or int, or 'auto', optional (default="auto")*) – Categorical features. If list of int, interpreted as indices. If list of strings, interpreted as feature names (need to specify `feature_name` as well). If 'auto' and data is pandas DataFrame, pandas categorical columns are used.
- **callbacks** (*list of callback functions or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **self** – Returns self.

Return type object

Note: Custom eval function expects a callable with following functions: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)`. Returns (eval_name, eval_result, is_bigger_better) or list of (eval_name, eval_result, is_bigger_better)

y_true: array-like of shape = [n_samples] The target values.

y_pred: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class)
The predicted values.

weight: array-like of shape = [n_samples] The weight of samples.

group: array-like Group/query data, used for ranking task.

eval_name: str The name of evaluation.

eval_result: float The eval result.

is_bigger_better: bool Is eval result bigger better, e.g. AUC is bigger_better.

For multi-class task, the y_pred is group by class_id first, then group by row_id. If you want to get i-th row y_pred in j-th class, the access way is y_pred[j * num_data + i].

---

class `lightgbm.`**`LGBMRanker`**(*boosting_type='gbdt'*, *num_leaves=31*, *max_depth=-1*, *learning_rate=0.1*, *n_estimators=10*, *max_bin=255*, *subsample_for_bin=50000*, *objective=None*, *min_split_gain=0.0*, *min_child_weight=5*, *min_child_samples=10*, *subsample=1.0*, *subsample_freq=1*, *colsample_bytree=1.0*, *reg_alpha=0.0*, *reg_lambda=0.0*, *random_state=0*, *n_jobs=-1*, *silent=True*, ***kwargs*)

Bases: `lightgbm.sklearn.LGBMModel`

LightGBM ranker.

Construct a gradient boosting model.

Parameters

- **`boosting_type`** (*`string, optional (default="gbdt")`*) – 'gbdt', traditional Gradient Boosting Decision Tree. 'dart', Dropouts meet Multiple Additive Regression Trees. 'goss', Gradient-based One-Side Sampling. 'rf', Random Forest.
- **`num_leaves`** (*`int, optional (default=31)`*) – Maximum tree leaves for base learners.
- **`max_depth`** (*`int, optional (default=-1)`*) – Maximum tree depth for base learners, -1 means no limit.
- **`learning_rate`** (*`float, optional (default=0.1)`*) – Boosting learning rate.
- **`n_estimators`** (*`int, optional (default=10)`*) – Number of boosted trees to fit.
- **`max_bin`** (*`int, optional (default=255)`*) – Number of bucketed bin for feature values.
- **`subsample_for_bin`** (*`int, optional (default=50000)`*) – Number of samples for constructing bins.
- **`objective`** (*`string, callable or None, optional (default=None)`*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). default: 'binary' for LGBMClassifier, 'lambdarank' for LGBMRanker.
- **`min_split_gain`** (*`float, optional (default=0.)`*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **`min_child_weight`** (*`int, optional (default=5)`*) – Minimum sum of instance weight(hessian) needed in a child(leaf).
- **`min_child_samples`** (*`int, optional (default=10)`*) – Minimum number of data need in a child(leaf).

- **subsample** (*float, optional (default=1.)*) – Subsample ratio of the training instance.
- **subsample_freq** (*int, optional (default=1)*) – Frequence of subsample, <=0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, optional (default=0)*) – Random number seed.
- **n_jobs** (*int, optional (default=-1)*) – Number of parallel threads.
- **silent** (*bool, optional (default=True)*) – Whether to print messages while running boosting.
- ****kwargs** (*other parameters*) – Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

Note: **kwargs is not supported in sklearn, it may cause unexpected issues.

n_features_
int – The number of features of fitted model.

classes_
array of shape = [n_classes] – The class label array (only for classification problem).

n_classes_
int – The number of classes (only for classification problem).

best_score_
dict or None – The best score of fitted model.

best_iteration_
int or None – The best iteration of fitted model if `early_stopping_rounds` has been specified.

objective_
string or callable – The concrete objective used while fitting this model.

booster_
Booster – The underlying Booster of this model.

evals_result_
dict or None – The evaluation results if `early_stopping_rounds` has been specified.

feature_importances_
array of shape = [n_features] – The feature importances (the higher, the more important the feature).

Note: A custom objective function can be provided for the `objective` parameter. In this case, it should have the signature `objective(y_true, y_pred) -> grad, hess` or `objective(y_true, y_pred, group) -> grad, hess`:

y_true: array-like of shape = [n_samples] The target values.

y_pred: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The predicted values.

group: array-like Group/query data, used for ranking task.

grad: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The value of the gradient for each sample point.

hess: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)
The value of the second derivative for each sample point.

For multi-class task, the y_pred is group by class_id first, then group by row_id. If you want to get i-th row y_pred in j-th class, the access way is y_pred[j * num_data + i] and you should group grad and hess in this way as well.

---

fit (*X, y, sample_weight=None, init_score=None, group=None, eval_set=None, eval_names=None, eval_sample_weight=None, eval_init_score=None, eval_group=None, eval_metric='ndcg', eval_at=[1], early_stopping_rounds=None, verbose=True, feature_name='auto', categorical_feature='auto', callbacks=None*)
Build a gradient boosting model from the training set (X, y).

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – Input feature matrix.
- **y** (*array-like of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Weights of training data.
- **init_score** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Init score of training data.
- **group** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Group data of training data.
- **eval_set** (*list or None, optional (default=None)*) – A list of (X, y) tuple pairs to use as a validation sets for early-stopping.
- **eval_names** (*list of strings or None, optional (default=None)*) – Names of eval_set.
- **eval_sample_weight** (*list of arrays or None, optional (default=None)*) – Weights of eval data.
- **eval_init_score** (*list of arrays or None, optional (default=None)*) – Init score of eval data.
- **eval_group** (*list of arrays or None, optional (default=None)*) – Group data of eval data.
- **eval_metric** (*string, list of strings, callable or None, optional (default="ndcg")*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note for more details.
- **eval_at** (*list of int, optional (default=[1])*) – The evaluation positions of NDCG.
- **early_stopping_rounds** (*int or None, optional (default=None)*) – Activates early stopping. The model will train until the validation score stops improving.

Validation error needs to decrease at least every `early_stopping_rounds` round(s) to continue training.

- **`verbose`** (*`bool, optional (default=True)`*) – If True and an evaluation set is used, writes the evaluation progress.
- **`feature_name`** (*`list of strings or 'auto', optional (default="auto")`*) – Feature names. If 'auto' and data is pandas DataFrame, data columns names are used.
- **`categorical_feature`** (*`list of strings or int, or 'auto', optional (default="auto")`*) – Categorical features. If list of int, interpreted as indices. If list of strings, interpreted as feature names (need to specify `feature_name` as well). If 'auto' and data is pandas DataFrame, pandas categorical columns are used.
- **`callbacks`** (*`list of callback functions or None, optional (default=None)`*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **self** – Returns self.

Return type object

Note: Custom eval function expects a callable with following functions: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)`. Returns (eval_name, eval_result, is_bigger_better) or list of (eval_name, eval_result, is_bigger_better)

y_true: array-like of shape = [n_samples] The target values.

y_pred: array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class) The predicted values.

weight: array-like of shape = [n_samples] The weight of samples.

group: array-like Group/query data, used for ranking task.

eval_name: str The name of evaluation.

eval_result: float The eval result.

is_bigger_better: bool Is eval result bigger better, e.g. AUC is bigger_better.

For multi-class task, the y_pred is group by class_id first, then group by row_id. If you want to get i-th row y_pred in j-th class, the access way is y_pred[j * num_data + i].

Callbacks

`lightgbm.`**`early_stopping`**(*stopping_rounds*, *verbose=True*)

Create a callback that activates early stopping.

Note: Activates early stopping. Requires at least one validation data and one metric. If there's more than one, will check all of them.

Parameters

- **`stopping_rounds`** (*`int`*) – The possible number of rounds without the trend occurrence.

- **verbose** (*bool, optional (default=True)*) – Whether to print message with early stopping information.

Returns **callback** – The callback that activates early stopping.

Return type function

lightgbm.**print_evaluation**(*period=1*, *show_stdv=True*)

Create a callback that prints the evaluation results.

Parameters

- **period** (*int, optional (default=1)*) – The period to print the evaluation results.
- **show_stdv** (*bool, optional (default=True)*) – Whether to show stdv (if provided).

Returns **callback** – The callback that prints the evaluation results every `period` iteration(s).

Return type function

lightgbm.**record_evaluation**(*eval_result*)

Create a callback that records the evaluation history into `eval_result`.

Parameters **eval_result** (*dict*) – A dictionary to store the evaluation results.

Returns **callback** – The callback that records the evaluation history into the passed dictionary.

Return type function

lightgbm.**reset_parameter**(***kwargs*)

Create a callback that resets the parameter after the first iteration.

Note: The initial parameter will still take in-effect on first iteration.

Parameters ****kwargs** (*value should be list or function*) – List of parameters for each boosting round or a customized function that calculates the parameter in terms of current number of round (e.g. yields learning rate decay). If list lst, parameter = lst[current_round]. If function func, parameter = func(current_round).

Returns **callback** – The callback that resets the parameter after the first iteration.

Return type function

Plotting

lightgbm.**plot_importance**(*booster*, *ax=None*, *height=0.2*, *xlim=None*, *ylim=None*, *title='Feature importance'*, *xlabel='Feature importance'*, *ylabel='Features'*, *importance_type='split'*, *max_num_features=None*, *ignore_zero=True*, *figsize=None*, *grid=True*, ***kwargs*)

Plot model's feature importances.

Parameters

- **booster** (Booster *or* LGBMModel) – Booster or LGBMModel instance which feature importance should be plotted.
- **ax** (*matplotlib.axes.Axes or None, optional (default=None)*) – Target axes instance. If None, new figure and axes will be created.

- **height** (*float, optional (default=0.2)*) – Bar height, passed to `ax.barh()`.
- **xlim** (*tuple of 2 elements or None, optional (default=None)*) – Tuple passed to `ax.xlim()`.
- **ylim** (*tuple of 2 elements or None, optional (default=None)*) – Tuple passed to `ax.ylim()`.
- **title** (*string or None, optional (default="Feature importance")*) – Axes title. If None, title is disabled.
- **xlabel** (*string or None, optional (default="Feature importance")*) – X-axis title label. If None, title is disabled.
- **ylabel** (*string or None, optional (default="Features")*) – Y-axis title label. If None, title is disabled.
- **importance_type** (*string, optional (default="split")*) – How the importance is calculated. If "split", result contains numbers of times the feature is used in a model. If "gain", result contains total gains of splits which use the feature.
- **max_num_features** (*int or None, optional (default=None)*) – Max number of top features displayed on plot. If None or <1, all features will be displayed.
- **ignore_zero** (*bool, optional (default=True)*) – Whether to ignore features with zero importance.
- **figsize** (*tuple of 2 elements or None, optional (default=None)*) – Figure size.
- **grid** (*bool, optional (default=True)*) – Whether to add a grid for axes.
- ****kwargs** (*other parameters*) – Other parameters passed to `ax.barh()`.

Returns **ax** – The plot with model's feature importances.

Return type matplotlib.axes.Axes

`lightgbm.`**`plot_metric`**(*booster*, *metric=None*, *dataset_names=None*, *ax=None*, *xlim=None*, *ylim=None*, *title='Metric during training'*, *xlabel='Iterations'*, *ylabel='auto'*, *figsize=None*, *grid=True*)

Plot one metric during training.

Parameters

- **booster** (*dict or* LGBMModel) – Dictionary returned from `lightgbm.train()` or LGBMModel instance.
- **metric** (*string or None, optional (default=None)*) – The metric name to plot. Only one metric supported because different metrics have various scales. If None, first metric picked from dictionary (according to hashcode).
- **dataset_names** (*list of strings or None, optional (default=None)*) – List of the dataset names which are used to calculate metric to plot. If None, all datasets are used.
- **ax** (*matplotlib.axes.Axes or None, optional (default=None)*) – Target axes instance. If None, new figure and axes will be created.
- **xlim** (*tuple of 2 elements or None, optional (default=None)*) – Tuple passed to `ax.xlim()`.
- **ylim** (*tuple of 2 elements or None, optional (default=None)*) – Tuple passed to `ax.ylim()`.

- **title** (*string or None, optional (default="Metric during training")*) – Axes title. If None, title is disabled.
- **xlabel** (*string or None, optional (default="Iterations")*) – X-axis title label. If None, title is disabled.
- **ylabel** (*string or None, optional (default="auto")*) – Y-axis title label. If 'auto', metric name is used. If None, title is disabled.
- **figsize** (*tuple of 2 elements or None, optional (default=None)*) – Figure size.
- **grid** (*bool, optional (default=True)*) – Whether to add a grid for axes.

Returns **ax** – The plot with metric's history over the training.

Return type matplotlib.axes.Axes

`lightgbm.`**`plot_tree`**(*booster*, *ax=None*, *tree_index=0*, *figsize=None*, *graph_attr=None*, *node_attr=None*, *edge_attr=None*, *show_info=None*)

Plot specified tree.

Parameters

- **booster** (Booster *or* LGBMModel) – Booster or LGBMModel instance to be plotted.
- **ax** (*matplotlib.axes.Axes or None, optional (default=None)*) – Target axes instance. If None, new figure and axes will be created.
- **tree_index** (*int, optional (default=0)*) – The index of a target tree to plot.
- **figsize** (*tuple of 2 elements or None, optional (default=None)*) – Figure size.
- **graph_attr** (*dict or None, optional (default=None)*) – Mapping of (attribute, value) pairs set for the graph.
- **node_attr** (*dict or None, optional (default=None)*) – Mapping of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict or None, optional (default=None)*) – Mapping of (attribute, value) pairs set for all edges.
- **show_info** (*list or None, optional (default=None)*) – What information should be showed on nodes. Possible values of list items: 'split_gain', 'internal_value', 'internal_count', 'leaf_count'.

Returns **ax** – The plot with single tree.

Return type matplotlib.axes.Axes

`lightgbm.`**`create_tree_digraph`**(*booster*, *tree_index=0*, *show_info=None*, *name=None*, *comment=None*, *filename=None*, *directory=None*, *format=None*, *engine=None*, *encoding=None*, *graph_attr=None*, *node_attr=None*, *edge_attr=None*, *body=None*, *strict=False*)

Create a digraph representation of specified tree.

Note: For more information please visit http://graphviz.readthedocs.io/en/stable/api.html#digraph.

Parameters

- **booster** (Booster *or* LGBMModel) – Booster or LGBMModel instance.

- **tree_index** (*int, optional (default=0)*) – The index of a target tree to convert.
- **show_info** (*list or None, optional (default=None)*) – What information should be showed on nodes. Possible values of list items: 'split_gain', 'internal_value', 'internal_count', 'leaf_count'.
- **name** (*string or None, optional (default=None)*) – Graph name used in the source code.
- **comment** (*string or None, optional (default=None)*) – Comment added to the first line of the source.
- **filename** (*string or None, optional (default=None)*) – Filename for saving the source. If None, `name` + '.gv' is used.
- **directory** (*string or None, optional (default=None)*) – (Sub)directory for source saving and rendering.
- **format** (*string or None, optional (default=None)*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*string or None, optional (default=None)*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*string or None, optional (default=None)*) – Encoding for saving the source.
- **graph_attr** (*dict or None, optional (default=None)*) – Mapping of (attribute, value) pairs set for the graph.
- **node_attr** (*dict or None, optional (default=None)*) – Mapping of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict or None, optional (default=None)*) – Mapping of (attribute, value) pairs set for all edges.
- **body** (*list of strings or None, optional (default=None)*) – Lines to add to the graph body.
- **strict** (*bool, optional (default=False)*) – Whether rendering should merge multi-edges.

Returns **graph** – The digraph representation of specified tree.

Return type graphviz.Digraph

CHAPTER 9

Parallel Learning Guide

This is a guide for parallel learning of LightGBM.

Follow the Quick Start to know how to use LightGBM first.

Choose Appropriate Parallel Algorithm

LightGBM provides 2 parallel learning algorithms now.

Parallel Algorithm	How to Use
Data parallel	`tree_learner=data`
Feature parallel	`tree_learner=feature`
Voting parallel	`tree_learner=voting`

These algorithms are suited for different scenarios, which is listed in the following table:

	#data is small	#data is large
#feature is small	Feature Parallel	Data Parallel
#feature is large	Feature Parallel	Voting Parallel

More details about these parallel algorithms can be found in optimization in parallel learning.

Build Parallel Version

Default build version support parallel learning based on the socket.

If you need to build parallel version with MPI support, please refer to Installation Guide.

Preparation

Socket Version

It needs to collect IP of all machines that want to run parallel learning in and allocate one TCP port (assume 12345 here) for all machines, and change firewall rules to allow income of this port (12345). Then write these IP and ports in one file (assume `mlist.txt`), like following:

```
machine1_ip 12345
machine2_ip 12345
```

MPI Version

It needs to collect IP (or hostname) of all machines that want to run parallel learning in. Then write these IP in one file (assume `mlist.txt`) like following:

```
machine1_ip
machine2_ip
```

Note: For Windows users, need to start "smpd" to start MPI service. More details can be found here.

Run Parallel Learning

Socket Version

1. Edit following parameters in config file:

`tree_learner=your_parallel_algorithm`, edit `your_parallel_algorithm` (e.g. feature/data) here.

`num_machines=your_num_machines`, edit `your_num_machines` (e.g. 4) here.

`machine_list_file=mlist.txt`, `mlist.txt` is created in *Preparation section*.

`local_listen_port=12345`, `12345` is allocated in *Preparation section*.

2. Copy data file, executable file, config file and `mlist.txt` to all machines.
3. Run following command on all machines, you need to change `your_config_file` to real config file.

For Windows: `lightgbm.exe config=your_config_file`

For Linux: `./lightgbm config=your_config_file`

MPI Version

1. Edit following parameters in config file:

`tree_learner=your_parallel_algorithm`, edit `your_parallel_algorithm` (e.g. feature/data) here.

`num_machines=your_num_machines`, edit `your_num_machines` (e.g. 4) here.

2. Copy data file, executable file, config file and `mlist.txt` to all machines. Note: MPI needs to be run in the **same path on all machines**.

3. Run following command on one machine (not need to run on all machines), need to change `your_config_file` to real config file.

For Windows: `mpiexec.exe /machinefile mlist.txt lightgbm.exe config=your_config_file`

For Linux: `mpiexec --machinefile mlist.txt ./lightgbm config=your_config_file`

Example

- A simple parallel example.

CHAPTER 10

LightGBM GPU Tutorial

The purpose of this document is to give you a quick step-by-step tutorial on GPU training.

For Windows, please see GPU Windows Tutorial.

We will use the GPU instance on Microsoft Azure cloud computing platform for demonstration, but you can use any machine with modern AMD or NVIDIA GPUs.

GPU Setup

You need to launch a `NV` type instance on Azure (available in East US, North Central US, South Central US, West Europe and Southeast Asia zones) and select Ubuntu 16.04 LTS as the operating system.

For testing, the smallest `NV6` type virtual machine is sufficient, which includes 1/2 M60 GPU, with 8 GB memory, 180 GB/s memory bandwidth and 4,825 GFLOPS peak computation power. Don't use the `NC` type instance as the GPUs (K80) are based on an older architecture (Kepler).

First we need to install minimal NVIDIA drivers and OpenCL development environment:

```
sudo apt-get update
sudo apt-get install --no-install-recommends nvidia-375
sudo apt-get install --no-install-recommends nvidia-opencl-icd-375 nvidia-opencl-dev␣
↪opencl-headers
```

After installing the drivers you need to restart the server.

```
sudo init 6
```

After about 30 seconds, the server should be up again.

If you are using a AMD GPU, you should download and install the AMDGPU-Pro driver and also install package `ocl-icd-libopencl1` and `ocl-icd-opencl-dev`.

Build LightGBM

Now install necessary building tools and dependencies:

```
sudo apt-get install --no-install-recommends git cmake build-essential libboost-dev␣
↪libboost-system-dev libboost-filesystem-dev
```

The NV6 GPU instance has a 320 GB ultra-fast SSD mounted at /mnt. Let's use it as our workspace (skip this if you are using your own machine):

```
sudo mkdir -p /mnt/workspace
sudo chown $(whoami):$(whoami) /mnt/workspace
cd /mnt/workspace
```

Now we are ready to checkout LightGBM and compile it with GPU support:

```
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build ; cd build
cmake -DUSE_GPU=1 ..
make -j$(nproc)
cd ..
```

You will see two binaries are generated, `lightgbm` and `lib_lightgbm.so`.

If you are building on OSX, you probably need to remove macro `BOOST_COMPUTE_USE_OFFLINE_CACHE` in `src/treelearner/gpu_tree_learner.h` to avoid a known crash bug in Boost.Compute.

Install Python Interface (optional)

If you want to use the Python interface of LightGBM, you can install it now (along with some necessary Python-package dependencies):

```
sudo apt-get -y install python-pip
sudo -H pip install setuptools numpy scipy scikit-learn -U
cd python-package/
sudo python setup.py install
cd ..
```

You need to set an additional parameter `"device" : "gpu"` (along with your other options like `learning_rate`, `num_leaves`, etc) to use GPU in Python.

You can read our Python Guide for more information on how to use the Python interface.

Dataset Preparation

Using the following commands to prepare the Higgs dataset:

```
git clone https://github.com/guolinke/boosting_tree_benchmarks.git
cd boosting_tree_benchmarks/data
wget "https://archive.ics.uci.edu/ml/machine-learning-databases/00280/HIGGS.csv.gz"
gunzip HIGGS.csv.gz
python higgs2libsvm.py
cd ../..
```

```
ln -s boosting_tree_benchmarks/data/higgs.train
ln -s boosting_tree_benchmarks/data/higgs.test
```

Now we create a configuration file for LightGBM by running the following commands (please copy the entire block and run it as a whole):

```
cat > lightgbm_gpu.conf <<EOF
max_bin = 63
num_leaves = 255
num_iterations = 50
learning_rate = 0.1
tree_learner = serial
task = train
is_training_metric = false
min_data_in_leaf = 1
min_sum_hessian_in_leaf = 100
ndcg_eval_at = 1,3,5,10
sparse_threshold = 1.0
device = gpu
gpu_platform_id = 0
gpu_device_id = 0
EOF
echo "num_threads=$(nproc)" >> lightgbm_gpu.conf
```

GPU is enabled in the configuration file we just created by setting `device=gpu`. It will use the first GPU installed on the system by default (`gpu_platform_id=0` and `gpu_device_id=0`).

Run Your First Learning Task on GPU

Now we are ready to start GPU training! First we want to verify the GPU works correctly. Run the following command to train on GPU, and take a note of the AUC after 50 iterations:

```
./lightgbm config=lightgbm_gpu.conf data=higgs.train valid=higgs.test␣
↪objective=binary metric=auc
```

Now train the same dataset on CPU using the following command. You should observe a similar AUC:

```
./lightgbm config=lightgbm_gpu.conf data=higgs.train valid=higgs.test␣
↪objective=binary metric=auc device=cpu
```

Now we can make a speed test on GPU without calculating AUC after each iteration.

```
./lightgbm config=lightgbm_gpu.conf data=higgs.train objective=binary metric=auc
```

Speed test on CPU:

```
./lightgbm config=lightgbm_gpu.conf data=higgs.train objective=binary metric=auc␣
↪device=cpu
```

You should observe over three times speedup on this GPU.

The GPU acceleration can be used on other tasks/metrics (regression, multi-class classification, ranking, etc) as well. For example, we can train the Higgs dataset on GPU as a regression task:

```
./lightgbm config=lightgbm_gpu.conf data=higgs.train objective=regression_l2 metric=l2
```

Also, you can compare the training speed with CPU:

```
./lightgbm config=lightgbm_gpu.conf data=higgs.train objective=regression_l2␣
↪metric=l2 device=cpu
```

Further Reading

GPU Tuning Guide and Performance Comparison

GPU SDK Correspondence and Device Targeting Table

GPU Windows Tutorial

Reference

Please kindly cite the following article in your publications if you find the GPU acceleration useful:

Huan Zhang, Si Si and Cho-Jui Hsieh. GPU Acceleration for Large-scale Tree Boosting. arXiv:1706.08359, 2017.

CHAPTER 11

Advanced Topics

Missing Value Handle

- LightGBM enables the missing value handle by default, you can disable it by set `use_missing=false`.
- LightGBM uses NA (NAN) to represent the missing value by default, you can change it to use zero by set `zero_as_missing=true`.
- When `zero_as_missing=false` (default), the unshown value in sparse matrices (and LightSVM) is treated as zeros.
- When `zero_as_missing=true`, NA and zeros (including unshown value in sparse matrices (and LightSVM)) are treated as missing.

Categorical Feature Support

- LightGBM can offer a good accuracy when using native categorical features. Not like simply one-hot coding, LightGBM can find the optimal split of categorical features. Such an optimal split can provide the much better accuracy than one-hot coding solution.
- Use `categorical_feature` to specify the categorical features. Refer to the parameter `categorical_feature` in *Parameters*.
- Converting to `int` type is needed first, and there is support for non-negative numbers only. It is better to convert into continues ranges.
- Use `max_cat_group`, `cat_smooth_ratio` to deal with over-fitting (when #data is small or #category is large).
- For categorical features with high cardinality (#category is large), it is better to convert it to numerical features.

LambdaRank

- The label should be `int` type, and larger numbers represent the higher relevance (e.g. 0:bad, 1:fair, 2:good, 3:perfect).
- Use `label_gain` to set the gain(weight) of `int` label.
- Use `max_position` to set the NDCG optimization position.

Parameters Tuning

- Refer to *Parameters Tuning*.

GPU Support

- Refer to *GPU Tutorial* and GPU Targets.

Parallel Learning

- Refer to *Parallel Learning Guide*.

CHAPTER 12

LightGBM FAQ

Contents

Critical

You encountered a critical issue when using LightGBM (crash, prediction error, non sense outputs...). Who should you contact?

If your issue is not critical, just post an issue in Microsoft/LightGBM repository.

If it is a critical issue, identify first what error you have:

- Do you think it is reproducible on CLI (command line interface), R, and/or Python?
- Is it specific to a wrapper? (R or Python?)
- Is it specific to the compiler? (gcc versions? MinGW versions?)
- Is it specific to your Operating System? (Windows? Linux?)
- Are you able to reproduce this issue with a simple case?
- Are you able to (not) reproduce this issue after removing all optimization flags and compiling LightGBM in debug mode?

Depending on the answers, while opening your issue, feel free to ping (just mention them with the arobase (@) symbol) appropriately so we can attempt to solve your problem faster:

- @guolinke (C++ code / R-package / Python-package)
- @Laurae2 (R-package)
- @wxchan (Python-package)
- @henry0312 (Python-package)
- @StrikerRUS (Python-package)
- @huanzhang12 (GPU support)

Remember this is a free/open community support. We may not be available 24/7 to provide support.

LightGBM

- **Question 1**: Where do I find more details about LightGBM parameters?
- **Solution 1**: Look at *Parameters* and Laurae++/Parameters website.

- **Question 2**: On datasets with million of features, training do not start (or starts after a very long time).
- **Solution 2**: Use a smaller value for `bin_construct_sample_cnt` and a larger value for `min_data`.

- **Question 3**: When running LightGBM on a large dataset, my computer runs out of RAM.
- **Solution 3**: Multiple solutions: set `histogram_pool_size` parameter to the MB you want to use for LightGBM (histogram_pool_size + dataset size = approximately RAM used), lower `num_leaves` or lower `max_bin` (see Microsoft/LightGBM#562).

- **Question 4**: I am using Windows. Should I use Visual Studio or MinGW for compiling LightGBM?
- **Solution 4**: It is recommended to use Visual Studio as its performance is higher for LightGBM.

- **Question 5**: When using LightGBM GPU, I cannot reproduce results over several runs.
- **Solution 5**: It is a normal issue, there is nothing we/you can do about, you may try to use `gpu_use_dp = true` for reproducibility (see Microsoft/LightGBM#560). You may also use CPU version.

- **Question 6**: Bagging is not reproducible when changing the number of threads.
- **Solution 6**: As LightGBM bagging is running multithreaded, its output is dependent on the number of threads used. There is no workaround currently.

- **Question 7**: I tried to use Random Forest mode, and LightGBM crashes!
- **Solution 7**: It is by design. You must use `bagging_fraction` and `feature_fraction` different from 1, along with a `bagging_freq`. See this thread as an example.

- **Question 8**: CPU are not kept busy (like 10% CPU usage only) in Windows when using LightGBM on very large datasets with many core systems.
- **Solution 8**: Please use Visual Studio as it may be 10x faster than MinGW especially for very large trees.

R-package

- **Question 1**: Any training command using LightGBM does not work after an error occurred during the training of a previous LightGBM model.
- **Solution 1**: Run `lgb.unloader(wipe = TRUE)` in the R console, and recreate the LightGBM datasets (this will wipe all LightGBM-related variables). Due to the pointers, choosing to not wipe variables will not fix the error. This is a known issue: Microsoft/LightGBM#698.
- **Question 2**: I used `setinfo`, tried to print my `lgb.Dataset`, and now the R console froze!
- **Solution 2**: Avoid printing the `lgb.Dataset` after using `setinfo`. This is a known bug: Microsoft/LightGBM#539.

Python-package

- **Question 1**: I see error messages like this when install from github using `python setup.py install`.

```
error: Error: setup script specifies an absolute path:

/Users/Microsoft/LightGBM/python-package/lightgbm/../../lib_lightgbm.so

setup() arguments must *always* be /-separated paths relative to the
setup.py directory, *never* absolute paths.
```

- **Solution 1**: this error should be solved in latest version. If you still meet this error, try to remove lightgbm.egg-info folder in your Python-package and reinstall, or check this thread on stackoverflow.

- **Question 2**: I see error messages like

```
Cannot get/set label/weight/init_score/group/num_data/num_feature before␣
↪construct dataset
```

but I already construct dataset by some code like

```
train = lightgbm.Dataset(X_train, y_train)
```

or error messages like

```
Cannot set predictor/reference/categorical feature after freed raw data, set free_
↪raw_data=False when construct Dataset to avoid this.
```

- **Solution 2**: Because LightGBM constructs bin mappers to build trees, and train and valid Datasets within one Booster share the same bin mappers, categorical features and feature names etc., the Dataset objects are constructed when construct a Booster. And if you set `free_raw_data=True` (default), the raw data (with Python data struct) will be freed. So, if you want to:

- get label(or weight/init_score/group) before construct dataset, it's same as get `self.label`
- set label(or weight/init_score/group) before construct dataset, it's same as `self.label=some_label_array`
- get num_data(or num_feature) before construct dataset, you can get data with `self.data`, then if your data is `numpy.ndarray`, use some code like `self.data.shape`
- set predictor(or reference/categorical feature) after construct dataset, you should set `free_raw_data=False` or init a Dataset object with the same raw data

CHAPTER 13

Development Guide

Algorithms

Refer to Features to understand important algorithms used in LightGBM.

Classes and Code Structure

Important Classes

Class	Description
`Application`	The entrance of application, including training and prediction logic
`Bin`	Data structure used for store feature discrete values(converted from float values)
`Boosting`	Boosting interface, current implementation is GBDT and DART
`Config`	Store parameters and configurations
`Dataset`	Store information of dataset
`DatasetLoader`	Used to construct dataset
`Feature`	Store One column feature
`Metric`	Evaluation metrics
`Network`	Network interfaces and communication algorithms
`ObjectiveFunction`	Objective function used to train
`Tree`	Store information of tree model
`TreeLearner`	Used to learn trees

Code Structure

Path	Description
./include	Header files
./include/utils	Some common functions
./src/application	Implementations of training and prediction logic
./src/boosting	Implementations of Boosting
./src/io	Implementations of IO relatived classes, including `Bin`, `Config`, `Dataset`, `DatasetLoader`, `Feature` and `Tree`
./src/metric	Implementations of metrics
./src/network	Implementations of network functions
./src/objective	Implementations of objective functions
./src/treelearner	Implementations of tree learners

Documents API

LightGBM support use doxygen to generate documents for classes and functions.

Refer to docs README.

C API

Refere to the comments in c_api.h.

High Level Language Package

See the implementations at Python-package and R-package.

Questions

Refer to FAQ.

Also feel free to open issues if you met problems.

CHAPTER 14

Indices and Tables

- genindex

Index