LightGBM Documentation

Release

Microsoft Corporation

May 07, 2017

Contents:

CHAPTER 1

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 guide*
- Python API Reference

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

For the detailed usage, please refer to Configuration.

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`,`binary`,`lambdarank`,`multiclass`, alias=`objective`,`app`
  - `regression`, regression application
  - `binary`, binary classification application
  - `lambdarank`, lambdarank application
  - `multiclass`, multi-class classification application, should set `num_class` as well
- `boosting`, default=`gbdt`, type=enum, options=`gbdt`,`dart`, alias=`boost`,`boosting_type`
  - `gbdt`, traditional Gradient Boosting Decision Tree
  - `dart`, Dropouts meet Multiple Additive Regression Trees
- `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 2

Python Package Introduction

This document gives a basic walkthrough of LightGBM python package.

List of other Helpful Links

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

Install

- Install the library first, follow the wiki here.
- Install python-package dependencies, `setuptools`, `numpy` and `scipy` is required, `scikit-learn` is required for sklearn interface and recommended. Run:

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

- In the `python-package` directory, run

```
python setup.py install
```

- 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 as follows:

```
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 = 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 3

Parameters

This is a page contains all parameters in LightGBM.

List of other Helpful Links

- Python API Reference
- *Parameters Tuning*

Update of 04/13/2017

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.

- `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
  - `multiclass`, multi-class classification application, should set `num_class` as well
- `boosting`, default=`gbdt`, type=enum, options=`gbdt`,`dart`, alias=`boost`,`boosting_type`
  - `gbdt`, traditional Gradient Boosting Decision Tree
  - `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: `num_tree` here equal with `num_iterations`. For multi-class, it actually learns `num_class * num_iterations` trees.
  - note: For python/R package, cannot use this parameters to control number of iterations.
- `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).
  - 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
  - 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

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`.
- `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. 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

- `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

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.
- `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).

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 4

Parameters Tuning

This is a page contains all parameters in LightGBM.

List of other Helpful Links

- *Parameters*
- Python API Reference

Convert parameters from XGBoost

LightGBM uses leaf-wise tree growth algorithm. But other popular tools, e.g. XGBoost, use depth-wise tree growth. So LightGBM use `num_leaves` to control complexity of tree model, and other tools usually use `max_depth`. Following table is the correspond between leaves and depths. The relation is `num_leaves = 2^(max_depth)`.

| max_depth | num_leaves | | ——— | ———- | | 1 | 2 | | 2 | 4 | | 3 | 8 | | 7 | 128 | | 10 | 1024 |

For faster speed

- Use bagging by set `bagging_fraction` and `bagging_freq`
- Use feature sub-sampling by set `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 5

lightgbm package

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.

`construct`()

Lazy init

`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/scipy.sparse*) – Data source of Dataset. When data type is string, it represents the path of txt file
- **`label`** (*list or numpy 1-D array, optional*) – Label of the training data.
- **`weight`** (*list or numpy 1-D array , optional*) – Weight for each instance.
- **`group`** (*list or numpy 1-D array , optional*) – Group/query size for dataset
- **`silent`** (*boolean, optional*) – Whether print messages during construction
- **`params`** (*dict, optional*) – Other parameters

`get_field`(*field_name*)

Get property from the Dataset.

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

Returns **info** – A numpy array of information of the data

Return type array

get_group()
Get the group of the Dataset.

Returns **init_score**

Return type array

get_init_score()
Get the initial score of the Dataset.

Returns **init_score**

Return type array

get_label()
Get the label of the Dataset.

Returns **label**

Return type array

get_weight()
Get the weight of the Dataset.

Returns **weight**

Return type array

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

Returns **number of rows**

Return type int

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

Returns **number of columns**

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 str*) – Name/index of categorical features

set_feature_name(*feature_name*)
Set feature name

Parameters **feature_name** (*list of str*) – Feature names

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

Parameters

- **field_name** (*str*) – The field name of the information
- **data** (*numpy array or list or None*) – The array ofdata to be set

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

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

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

Parameters **init_score** (*numpy array or list or None*) – Init score for booster

set_label(*label*)
Set label of Dataset

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

set_reference(*reference*)
Set reference dataset

Parameters **reference** (Dataset) – Will use reference as template to consturct current dataset

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

Parameters **weight** (*numpy array or list or None*) – Weight for each data point

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

Parameters

- **used_indices** (*list of int*) – Used indices of this subset
- **params** (*dict*) – Other parameters

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

"Booster in LightGBM.

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

Parameters

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

attr(*key*)
Get attribute string from the Booster.

Parameters **key** (*str*) – The key to get attribute from.

Returns **value** – The attribute value of the key, returns None if attribute do not exist.

Return type str

dump_model(*num_iteration=-1*)
Dump model to json format

Parameters **num_iteration** (*int*) – Number of iteration that want to dump. < 0 means dump to best iteration(if have)

Returns

Return type Json format of model

eval(*data*, *name*, *feval=None*)
Evaluate for data

Parameters

- **data** (*Dataset object*) –
- **name** – Name of data
- **feval** (*function*) – Custom evaluation function.

Returns **result** – Evaluation result list.

Return type list

eval_train(*feval=None*)
Evaluate for training data

Parameters **feval** (*function*) – Custom evaluation function.

Returns **result** – Evaluation result list.

Return type str

eval_valid(*feval=None*)
Evaluate for validation data

Parameters **feval** (*function*) – Custom evaluation function.

Returns **result** – Evaluation result list.

Return type str

feature_importance(*importance_type='split'*)
Get feature importances

Parameters

- **importance_type** (*str, default "split"*) –
- **the importance is calculated** (*How*) –
- **is the number of times a feature is used in a model** (*"split"*) –
- **is the total gain of splits which use the feature** (*"gain"*) –

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

Return type array

feature_name()
Get feature names.

Returns **result** – Array of feature names.

Return type array

params_str = None
construct booster object

predict(*data*, *num_iteration=-1*, *raw_score=False*, *pred_leaf=False*, *data_has_header=False*, *is_reshape=True*)
Predict logic

Parameters

- **data** (*string/numpy array/scipy.sparse*) – Data source for prediction When data type is string, it represents the path of txt file
- **num_iteration** (*int*) – Used iteration for prediction, < 0 means predict for best iteration(if have)
- **raw_score** (*bool*) – True for predict raw score
- **pred_leaf** (*bool*) – True for predict leaf index
- **data_has_header** (*bool*) – Used for txt data
- **is_reshape** (*bool*) – Reshape to (nrow, ncol) if true

Returns

Return type Prediction result

reset_parameter(*params*)
Reset parameters for booster

Parameters

- **params** (*dict*) – New parameters for boosters
- **silent** (*boolean, optional*) – Whether print messages during construction

rollback_one_iter()
Rollback one iteration

save_model(*filename*, *num_iteration=-1*)
Save model of booster to file

Parameters

- **filename** (*str*) – Filename to save
- **num_iteration** (*int*) – Number of iteration that want to save. < 0 means save the best iteration(if have)

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

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

update(*train_set=None*, *fobj=None*)
Update for one iteration Note: 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

Parameters

- **train_set** – Training data, None means use last training data
- **fobj** (*function*) – Customized objective function.

Returns

Return type is_finished, 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*, *callbacks=None*)

Train with given parameters.

Parameters

- **params** (*dict*) – Parameters for training.
- **train_set** (Dataset) – Data to be trained.
- **num_boost_round** (*int*) – Number of boosting iterations.
- **valid_sets** (*list of Datasets*) – List of data to be evaluated during training
- **valid_names** (*list of string*) – Names of valid_sets
- **fobj** (*function*) – Customized objective function.
- **feval** (*function*) – Customized evaluation function. Note: should return (eval_name, eval_result, is_higher_better) of list of this
- **init_model** (*file name of lightgbm model or 'Booster' instance*) – model used for continued train
- **feature_name** (*list of str, or 'auto'*) – Feature names If 'auto' and data is pandas DataFrame, use data columns name
- **categorical_feature** (*list of str or int, or 'auto'*) – Categorical features, type int represents index, type str represents feature names (need to specify feature_name as well) If 'auto' and data is pandas DataFrame, use pandas categorical columns
- **early_stopping_rounds** (*int*) – Activates early stopping. Requires at least one validation data and one metric If there's more than one, will check all of them Returns the model with (best_iter + early_stopping_rounds) If early stopping occurs, the model will add 'best_iteration' field
- **evals_result** (*dict or None*) – This dictionary used to store all evaluation results of all the items in valid_sets. Example: with a valid_sets containing [valid_set, train_set]

  and valid_names containing ['eval', 'train'] and a paramater containing ('metric':'logloss')

  Returns: {'train': {'logloss': ['0.48253', '0.35953', ...]}, 'eval': {'logloss': ['0.480385', '0.357756', ...]}}

  passed with None means no using this function
- **verbose_eval** (*bool or int*) – Requires at least one item in evals. If *verbose_eval* is True,

  the eval metric on the valid set is printed at each boosting stage.

  If *verbose_eval* is 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 or function*) – List of learning rate 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) - list l: learning_rate = l[current_round] - function f: learning_rate = f(current_round)
- **callbacks** (*list of callback functions*) – List of callback functions that are applied at each iteration. See Callbacks in Python-API.md for more information.

Returns booster

Return type a trained booster model

lightgbm.**cv**(*params*, *train_set*, *num_boost_round=10*, *folds=None*, *nfold=5*, *stratified=False*, *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*)

Cross-validation with given paramaters.

Parameters

- **params** (*dict*) – Booster params.
- **train_set** (Dataset) – Data to be trained.
- **num_boost_round** (*int*) – Number of boosting iterations.
- **folds** (*a generator or iterator of (train_idx, test_idx) tuples*) – The train indices and test indices for each folds. This argument has highest priority over other data split arguments.
- **nfold** (*int*) – Number of folds in CV.
- **stratified** (*bool*) – Perform stratified sampling.
- **shuffle** (*bool*) – Whether shuffle before split data
- **metrics** (*string or list of strings*) – Evaluation metrics to be watched in CV. If *metrics* is not None, the metric in *params* will be overridden.
- **fobj** (*function*) – Custom objective function.
- **feval** (*function*) – Custom evaluation function.
- **init_model** (*file name of lightgbm model or 'Booster' instance*) – model used for continued train
- **feature_name** (*list of str, or 'auto'*) – Feature names If 'auto' and data is pandas DataFrame, use data columns name
- **categorical_feature** (*list of str or int, or 'auto'*) – Categorical features, type int represents index, type str represents feature names (need to specify feature_name as well) If 'auto' and data is pandas DataFrame, use pandas categorical columns
- **early_stopping_rounds** (*int*) – 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** (*function*) – Preprocessing function that takes (dtrain, dtest, param) and returns transformed versions of those.

- **verbose_eval** (*bool, int, or None, 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 boosting stage. If an integer is given,

  progress will be displayed at every given *verbose_eval* boosting stage.

- **show_stdv** (*bool, default True*) – Whether to display the standard deviation in progress. Results are not affected, and always contains std.
- **seed** (*int*) – Seed used to generate the folds (passed to numpy.random.seed).
- **callbacks** (*list of callback functions*) – List of callback functions that are applied at each iteration. See Callbacks in Python-API.md for more information.

Returns evaluation history

Return type list(string)

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='regression', min_split_gain=0, min_child_weight=5, min_child_samples=10, subsample=1, subsample_freq=1, colsample_bytree=1, reg_alpha=0, reg_lambda=0, scale_pos_weight=1, is_unbalance=False, seed=0, nthread=-1, silent=True, sigmoid=1.0, huber_delta=1.0, gaussian_eta=1.0, fair_c=1.0, poisson_max_delta_step=0.7, max_position=20, label_gain=None, drop_rate=0.1, skip_drop=0.5, max_drop=50, uniform_drop=False, xgboost_dart_mode=False*)

Bases: object

apply(*X, num_iteration=0*)

Return the predicted leaf every tree for each sample.

Parameters

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

Returns X_leaves

Return type array_like, shape=[n_samples, n_trees]

booster_

Get the underlying lightgbm Booster of this model.

evals_result_

Get the evaluation results.

feature_importances_

Get normailized feature importances.

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*)

Fit the gradient boosting model

Parameters

- **X** (*array_like*) – Feature matrix
- **y** (*array_like*) – Labels
- **sample_weight** (*array_like*) – weight of training data
- **init_score** (*array_like*) – init score of training data
- **group** (*array_like*) – group data of training data
- **eval_set** (*list, optional*) – A list of (X, y) tuple pairs to use as a validation set for early-stopping
- **eval_names** (*list of string*) – Names of eval_set
- **eval_sample_weight** (*List of array*) – weight of eval data
- **eval_init_score** (*List of array*) – init score of eval data
- **eval_group** (*List of array*) – group data of eval data
- **eval_metric** (*str, list of str, callable, optional*) – If a str, should be a built-in evaluation metric to use. If callable, a custom evaluation metric, see note for more details.
- **early_stopping_rounds** (*int*) –
- **verbose** (*bool*) – If *verbose* and an evaluation set is used, writes the evaluation
- **feature_name** (*list of str, or 'auto'*) – Feature names If 'auto' and data is pandas DataFrame, use data columns name
- **categorical_feature** (*list of str or int, or 'auto'*) – Categorical features, type int represents index, type str represents feature names (need to specify feature_name as well) If 'auto' and data is pandas DataFrame, use pandas categorical columns
- **callbacks** (*list of callback functions*) – List of callback functions that are applied at each iteration. See Callbacks in Python-API.md for more information.

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)`.

return (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_class] (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 name of evaluation

eval_result: float 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]

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

Parameters

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

Returns predicted_result

Return type array_like, shape=[n_samples] or [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='binary'*, *min_split_gain=0*, *min_child_weight=5*, *min_child_samples=10*, *subsample=1*, *subsample_freq=1*, *colsample_bytree=1*, *reg_alpha=0*, *reg_lambda=0*, *scale_pos_weight=1*, *is_unbalance=False*, *seed=0*, *nthread=-1*, *silent=True*, *sigmoid=1.0*, *drop_rate=0.1*, *skip_drop=0.5*, *max_drop=50*, *uniform_drop=False*, *xgboost_dart_mode=False*)
Bases: lightgbm.sklearn.LGBMModel, object

classes_
Get class label array.

n_classes_
Get 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, shape=[n_samples, n_features]*) – Input features matrix.
- **num_iteration** (*int*) – Limit number of iterations in the prediction; defaults to 0 (use all trees).

Returns predicted_probability

Return type array_like, 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='regression'*, *min_split_gain=0*, *min_child_weight=5*, *min_child_samples=10*, *subsample=1*, *subsample_freq=1*, *colsample_bytree=1*, *reg_alpha=0*, *reg_lambda=0*, *seed=0*, *nthread=-1*, *silent=True*, *huber_delta=1.0*, *gaussian_eta=1.0*, *fair_c=1.0*, *poisson_max_delta_step=0.7*, *drop_rate=0.1*, *skip_drop=0.5*, *max_drop=50*, *uniform_drop=False*, *xgboost_dart_mode=False*)
Bases: lightgbm.sklearn.LGBMModel, object

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='lambdarank'*, *min_split_gain=0*, *min_child_weight=5*, *min_child_samples=10*, *subsample=1*, *subsample_freq=1*, *colsample_bytree=1*, *reg_alpha=0*, *reg_lambda=0*, *scale_pos_weight=1*, *is_unbalance=False*, *seed=0*, *nthread=-1*, *silent=True*, *sigmoid=1.0*, *max_position=20*, *label_gain=None*, *drop_rate=0.1*, *skip_drop=0.5*, *max_drop=50*, *uniform_drop=False*, *xgboost_dart_mode=False*)

Bases: lightgbm.sklearn.LGBMModel

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*)

Most arguments like common methods except following:

eval_at [list of int] The evaulation positions of NDCG

Callbacks

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

Create a callback that activates early stopping. 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 stopping rounds before the trend occur.
- **verbose** (*optional, bool*) – Whether to print message about early stopping information.

Returns **callback** – The requested callback function.

Return type function

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

Create a callback that print evaluation result.

Parameters

- **period** (*int*) – The period to log the evaluation results
- **show_stdv** (*bool, optional*) – Whether show stdv if provided

Returns **callback** – A callback that print evaluation every period iterations.

Return type function

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

Create a call back that records the evaluation history into eval_result.

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

Returns **callback** – The requested callback function.

Return type function

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

Reset parameter after 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 learning_rate in terms of current number of round (e.g. yields learning rate decay) - list l: parameter = l[current_round] - function f: parameter = f(current_round)

Returns **callback** – The requested callback function.

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 feature importances.

Parameters

- **booster** (*Booster or LGBMModel*) – Booster or LGBMModel instance
- **ax** (*matplotlib Axes*) – Target axes instance. If None, new figure and axes will be created.
- **height** (*float*) – Bar height, passed to ax.barh()
- **xlim** (*tuple of 2 elements*) – Tuple passed to axes.xlim()
- **ylim** (*tuple of 2 elements*) – Tuple passed to axes.ylim()
- **title** (*str*) – Axes title. Pass None to disable.
- **xlabel** (*str*) – X axis title label. Pass None to disable.
- **ylabel** (*str*) – Y axis title label. Pass None to disable.
- **importance_type** (*str*) – How the importance is calculated: "split" or "gain" "split" is the number of times a feature is used in a model "gain" is the total gain of splits which use the feature
- **max_num_features** (*int*) – Max number of top features displayed on plot. If None or smaller than 1, all features will be displayed.
- **ignore_zero** (*bool*) – Ignore features with zero importance
- **figsize** (*tuple of 2 elements*) – Figure size
- **grid** (*bool*) – Whether add grid for axes
- ****kwargs** – Other keywords passed to ax.barh()

Returns **ax**

Return type matplotlib 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*) – Evals_result recorded by lightgbm.train() or LGBMModel instance

- **metric** (*str or None*) – The metric name to plot. Only one metric supported because different metrics have various scales. Pass None to pick *first* one (according to dict hashcode).
- **dataset_names** (*None or list of str*) – List of the dataset names to plot. Pass None to plot all datasets.
- **ax** (*matplotlib Axes*) – Target axes instance. If None, new figure and axes will be created.
- **xlim** (*tuple of 2 elements*) – Tuple passed to axes.xlim()
- **ylim** (*tuple of 2 elements*) – Tuple passed to axes.ylim()
- **title** (*str*) – Axes title. Pass None to disable.
- **xlabel** (*str*) – X axis title label. Pass None to disable.
- **ylabel** (*str*) – Y axis title label. Pass None to disable. Pass 'auto' to use *metric*.
- **figsize** (*tuple of 2 elements*) – Figure size
- **grid** (*bool*) – Whether add grid for axes

Returns ax

Return type matplotlib 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, LGBMModel) – Booster or LGBMModel instance.
- **ax** (*matplotlib Axes*) – Target axes instance. If None, new figure and axes will be created.
- **tree_index** (*int, default 0*) – Specify tree index of target tree.
- **figsize** (*tuple of 2 elements*) – Figure size.
- **graph_attr** (*dict*) – Mapping of (attribute, value) pairs for the graph.
- **node_attr** (*dict*) – Mapping of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict*) – Mapping of (attribute, value) pairs set for all edges.
- **show_info** (*list*) – Information shows on nodes. options: 'split_gain', 'internal_value', 'internal_count' or 'leaf_count'.

Returns ax

Return type matplotlib 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 of specified tree.

See:

- http://graphviz.readthedocs.io/en/stable/api.html#digraph

Parameters

- **booster** (*Booster, LGBMModel*) – Booster or LGBMModel instance.
- **tree_index** (*int, default 0*) – Specify tree index of target tree.
- **show_info** (*list*) – Information shows on nodes. options: 'split_gain', 'internal_value', 'internal_count' or 'leaf_count'.
- **name** (*str*) – Graph name used in the source code.
- **comment** (*str*) – Comment added to the first line of the source.
- **filename** (*str*) – Filename for saving the source (defaults to name + '.gv').
- **directory** (*str*) – (Sub)directory for source saving and rendering.
- **format** (*str*) – Rendering output format ('pdf', 'png', ...).
- **engine** (*str*) – Layout command used ('dot', 'neato', ...).
- **encoding** (*str*) – Encoding for saving the source.
- **graph_attr** (*dict*) – Mapping of (attribute, value) pairs for the graph.
- **node_attr** (*dict*) – Mapping of (attribute, value) pairs set for all nodes.
- **edge_attr** (*dict*) – Mapping of (attribute, value) pairs set for all edges.
- **body** (*list of str*) – Iterable of lines to add to the graph body.
- **strict** (*bool*) – Iterable of lines to add to the graph body.

Returns **graph**

Return type graphviz Digraph

CHAPTER 6

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_train_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

CHAPTER 7

LightGBM FAQ

Catalog

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 8

Development Guide

Algorithms

Refer to Features to get 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` | Newwork 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 |

API Documents

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

C API

Refere to the comments in c_api.h.

High level Language package

Follow the implementation of python-package.

Ask Questions

Feel free to open issues if you met problems.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index

T

U