CONTENTS:

1 Installation Guide 3

2 Quick Start 17

3 Python-package Introduction 19

4 Features 23

5 Experiments 31

6 Parameters 37

7 Parameters Tuning 59

8 C API 65

9 Python API 95

10 Distributed Learning Guide 169

11 LightGBM GPU Tutorial 179

12 Advanced Topics 183

13 LightGBM FAQ 185

14 Development Guide 193

15 GPU Tuning Guide and Performance Comparison 195

16 GPU SDK Correspondence and Device Targeting Table 199

17 GPU Windows Compilation 203

18 Recommendations When Using gcc 223

19 Documentation 227

20 Indices and Tables 229

Index 231
LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages:

- Faster training speed and higher efficiency.
- Lower memory usage.
- Better accuracy.
- Support of parallel, distributed, and GPU learning.
- Capable of handling large-scale data.

For more details, please refer to Features.
This is a guide for building the LightGBM Command Line Interface (CLI). If you want to build the Python-package or R-package please refer to Python-package and R-package folders respectively.

All instructions below are aimed at compiling the 64-bit version of LightGBM. It is worth compiling the 32-bit version only in very rare special cases involving environmental limitations. The 32-bit version is slow and untested, so use it at your own risk and don’t forget to adjust some of the commands below when installing.

If you need to build a static library instead of a shared one, you can add `-DBUILD_STATIC_LIB=ON` to CMake flags.

Users who want to perform benchmarking can make LightGBM output time costs for different internal routines by adding `-DUSE_TIMETAG=ON` to CMake flags.

It is possible to build LightGBM in debug mode. In this mode all compiler optimizations are disabled and LightGBM performs more checks internally. To enable debug mode you can add `-DUSE_DEBUG=ON` to CMake flags or choose Debug_* configuration (e.g. Debug_DLL, Debug_mpi) in Visual Studio depending on how you are building LightGBM.

You can also download the artifacts of the latest successful build on master branch (nightly builds) here:.

Contents

- Windows
- Linux
- macOS
- Docker
- Build Threadless Version (not Recommended)
- Build MPI Version
- Build GPU Version
- Build CUDA Version (Experimental)
- Build HDFS Version
- Build Java Wrapper
1.1 Windows

On Windows LightGBM can be built using

• Visual Studio;
• CMake and VS Build Tools;
• CMake and MinGW.

1.1.1 Visual Studio (or VS Build Tools)

With GUI

1. Install Visual Studio (2015 or newer).
2. Download zip archive and unzip it.
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 VS Build Tools (VS Build Tools is not needed if Visual Studio (2015 or newer) is already installed).
2. Run the following commands:

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

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

1.1.2 MinGW-w64

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 you encounter the sh.exe was found in your PATH error.

It is recommended that you use Visual Studio since it has better multithreading efficiency in Windows for many-core systems (see Question 4 and Question 8).
Also, you may want to read gcc Tips.

1.2 Linux

On Linux LightGBM can be built using CMake and gcc or Clang.

1. Install CMake.
2. Run the following commands:

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

Note: glibc >= 2.14 is required.
Also, you may want to read gcc Tips.

1.3 macOS

On macOS LightGBM can be built using CMake and Apple Clang or gcc.

1.3.1 Apple Clang

Only Apple Clang version 8.1 or higher is supported.

1. Install CMake (3.16 or higher):

```bash
brew install cmake
```
2. Install OpenMP:

```bash
brew install libomp
```
3. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake ..
make -j4
```
1.3.2 gcc

1. Install CMake (3.2 or higher):

   ```sh
   brew install cmake
   ```

2. Install gcc:

   ```sh
   brew install gcc
   ```

3. Run the following commands:

   ```sh
   git clone --recursive https://github.com/microsoft/LightGBM
   cd LightGBM
   export CXX=g++-7 CC=gcc-7  # replace "7" with version of gcc installed on your machine
   mkdir build
   cd build
   cmake ..
   make -j4
   ```

Also, you may want to read gcc Tips.

1.4 Docker

Refer to Docker folder.

1.5 Build Threadless Version (not Recommended)

The default build version of LightGBM is based on OpenMP. You can build LightGBM without OpenMP support but it is strongly not recommended.

1.5.1 Windows

On Windows a version of LightGBM without OpenMP support can be built using

- Visual Studio;
- CMake and VS Build Tools;
- CMake and MinGW.

**Visual Studio (or VS Build Tools)**

**With GUI**

1. Install Visual Studio (2015 or newer).
2. Download zip archive and unzip it.
5. Go to PROJECT -> Properties -> Configuration Properties -> C/C++ -> Language and change the OpenMP Support property to No (/openmp-).

6. Get back to the project’s main screen, then 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 VS Build Tools (VS Build Tools is not needed if Visual Studio (2015 or newer) is already installed).

2. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -A x64 -DUSE_OPENMP=OFF ..
cmake --build . --target ALL_BUILD --config Release
```

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

MinGW-w64

1. Install Git for Windows, CMake and MinGW-w64.

2. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -G "MinGW Makefiles" -DUSE_OPENMP=OFF ..
ingw32-make.exe -j4
```

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

Note: You may need to run the cmake -G "MinGW Makefiles" -DUSE_OPENMP=OFF .. one more time if you encounter the sh.exe was found in your PATH error.

1.5.2 Linux

On Linux a version of LightGBM without OpenMP support can be built using CMake and gcc or Clang.

1. Install CMake.

2. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
```

(continues on next page)
cd build
cmake -DUSE_OPENMP=OFF ..
make -j4

Note: glibc >= 2.14 is required.

### 1.5.3 macOS

On macOS a version of LightGBM without OpenMP support can be built using **CMake** and **Apple Clang** or **gcc**.

**Apple Clang**

Only **Apple Clang** version 8.1 or higher is supported.

1. Install **CMake** (3.16 or higher):

   ```
   brew install cmake
   ```

2. Run the following commands:

   ```
   git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -DUSE_OPENMP=OFF ..
make -j4
   ```

**gcc**

1. Install **CMake** (3.2 or higher):

   ```
   brew install cmake
   ```

2. **Install gcc**:

   ```
   brew install gcc
   ```

3. Run the following commands:

   ```
   git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=g++-7 CC=gcc-7 # replace "7" with version of gcc installed on your machine
mkdir build
cd build
cmake -DUSE_OPENMP=OFF ..
make -j4
   ```
1.6 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 distributed learning application with high performance communication, you can build the LightGBM with MPI support.

1.6.1 Windows

On Windows an MPI version of LightGBM can be built using

- MS MPI and Visual Studio;
- MS MPI, CMake and VS Build Tools.

With GUI

1. You need to install MS MPI first. Both msmpisdk.msi and msmpissetup.exe are needed.
2. Install Visual Studio (2015 or newer).
3. Download zip archive and unzip it.
5. 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 msmpissetup.exe are needed.
2. Install Git for Windows, CMake (3.8 or higher) and VS Build Tools (VS Build Tools is not needed if Visual Studio (2015 or newer) is already installed).
3. Run the following commands:

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

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

Note: Building MPI version by MinGW is not supported due to the miss of MPI library in it.
1.6.2 Linux

On Linux an MPI version of LightGBM can be built using Open MPI, CMake and gcc or Clang.

1. Install Open MPI.
2. Install CMake.
3. 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.

1.6.3 macOS

On macOS an MPI version of LightGBM can be built using Open MPI, CMake and Apple Clang or gcc.

**Apple Clang**

Only Apple Clang version 8.1 or higher is supported.

1. Install CMake (3.16 or higher):

```
brew install cmake
```

2. Install OpenMP:

```
brew install libomp
```

3. Install Open MPI:

```
brew install open-mpi
```

4. Run the following commands:

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

1. Install **CMake** (3.2 or higher):

   ```
   brew install cmake
   ```

2. Install **gcc**:

   ```
   brew install gcc
   ```

3. Install **Open MPI**:

   ```
   brew install open-mpi
   ```

4. Run the following commands:

   ```
   git clone --recursive https://github.com/microsoft/LightGBM
   cd LightGBM
   export CXX=g++-7 CC=gcc-7  # replace "7" with version of gcc installed on your machine
   mkdir build
   cd build
   cmake -DUSE_MPI=ON ..
   make -j4
   ```

### 1.7 Build GPU Version

#### 1.7.1 Linux

On Linux a GPU version of LightGBM (`device_type=gpu`) can be built using **OpenCL**, **Boost**, **CMake** and gcc or Clang.

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 `ocl-icd-libopencl1` and `ocl-icd-opencl-dev`) can also be used.

- **libboost** 1.56 or later (1.61 or later is 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
```
1.7.2 Windows

On Windows a GPU version of LightGBM (device_type=gpu) can be built using OpenCL, Boost, CMake and VS Build Tools or MinGW.

If you use MinGW, the build procedure is similar to the build on 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 VS Build Tools (VS Build Tools is not needed if Visual Studio (2015 or newer) 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.

Further reading and correspondence table: GPU SDK Correspondence and Device Targeting Table.

3. Install Boost Binaries.

   Note: Match your Visual C++ version:
   - Visual Studio 2015 -> msvc-14.0-64.exe,
   - Visual Studio 2017 -> msvc-14.1-64.exe,
   - Visual Studio 2019 -> msvc-14.2-64.exe.

4. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -A x64 -DUSE_GPU=1 -DBOOST_ROOT=C:/local/boost_1_63_0 -DBOOST_LIBRARYDIR=C:/local/boost_1_63_0/lib64-msvc-14.0 ..
# if you have installed NVIDIA CUDA to a customized location, you should specify paths to OpenCL headers and library like the following:
# cmake -A x64 -DUSE_GPU=1 -DOpenCL_LIBRARY=/usr/local/cuda/lib64/libOpenCL.so -DOpenCL_INCREASE_DIR=/usr/local/cuda/include/ ..
cmake --build . --target ALL_BUILD --config Release
```

Note: C:/local/boost_1_63_0 and C:/local/boost_1_63_0/lib64-msvc-14.0 are locations of your Boost binaries (assuming you’ve downloaded 1.63.0 version for Visual Studio 2015).
1.7.3 Docker

Refer to GPU Docker folder.

1.8 Build CUDA Version (Experimental)

The original GPU build of LightGBM (device_type=gpu) is based on OpenCL.

The CUDA-based build (device_type=cuda) is a separate implementation and requires an NVIDIA graphics card with compute capability 6.0 and higher. It should be considered experimental, and we suggest using it only when it is impossible to use OpenCL version (for example, on IBM POWER microprocessors).

Note: only Linux is supported, other operating systems are not supported yet.

1.8.1 Linux

On Linux a CUDA version of LightGBM can be built using CUDA, CMake and gcc or Clang.

The following dependencies should be installed before compilation:

- CUDA 9.0 or later libraries. Please refer to this detailed guide. Pay great attention to the minimum required versions of host compilers listed in the table from that guide and use only recommended versions of compilers.
- CMake 3.16 or later.

To build LightGBM CUDA version, run the following commands:

```
$ git clone --recursive https://github.com/microsoft/LightGBM
$ cd LightGBM
$ mkdir build
$ cd build
$ cmake -DUSE_CUDA=1 ..
$ make -j4
```

1.9 Build HDFS Version

The HDFS version of LightGBM was tested on CDH-5.14.4 cluster.

1.9.1 Linux

On Linux a HDFS version of LightGBM can be built using CMake and gcc.

1. Install CMake.

2. Run the following commands:

```
$ git clone --recursive https://github.com/microsoft/LightGBM
$ cd LightGBM
$ mkdir build
$ cd build
$ cmake -DUSE_HDFS=ON ..
$ make -j4
```

# if you have installed HDFS to a customized location, you should specify paths to HDFS headers (hdfs.h) and library (libhdfs.so) like the following:

(continues on next page)
1.10 Build Java Wrapper

Using the following instructions you can generate a JAR file containing the LightGBM C API wrapped by SWIG.

1.10.1 Windows

On Windows a Java wrapper of LightGBM can be built using Java, SWIG, CMake and VS Build Tools or MinGW.

VS Build Tools

1. Install Git for Windows, CMake (3.8 or higher) and VS Build Tools (VS Build Tools is not needed if Visual Studio (2015 or newer) is already installed).
2. Install SWIG and Java (also make sure that JAVA_HOME is set properly).
3. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
# cmake \n# -DUSE_HDFS=ON \n# -DHDFS_LIB="/opt/cloudera/parcels/CDH-5.14.4-1.cdh5.14.4.p0.3/lib64/libhdfs.so" \n# -DHDFS_INCLUDE_DIR="/opt/cloudera/parcels/CDH-5.14.4-1.cdh5.14.4.p0.3/include/" \n# ..
make -j4
```

The .jar file will be in LightGBM/build folder and the .dll files will be in LightGBM/Release folder.

MinGW-w64

1. Install Git for Windows, CMake and MinGW-w64.
2. Install SWIG and Java (also make sure that JAVA_HOME is set properly).
3. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -G "MinGW Makefiles" -DUSE_SWIG=ON ..
cmake --build . --target ALL_BUILD --config Release
```

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -A x64 -DUSE_SWIG=ON ..
cmake --build . --target ALL_BUILD --config Release
```
The .jar file will be in LightGBM/build folder and the .dll files will be in LightGBM/ folder.

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

It is recommended to use VS Build Tools (Visual Studio) since it has better multithreading efficiency in Windows for many-core systems (see Question 4 and Question 8).

Also, you may want to read gcc Tips.

### 1.10.2 Linux

On Linux a Java wrapper of LightGBM can be built using Java, SWIG, CMake and gcc or Clang.

1. Install CMake, SWIG and Java (also make sure that JAVA_HOME is set properly).
2. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -DUSE_SWIG=ON ..
make -j4
```

### 1.10.3 macOS

On macOS a Java wrapper of LightGBM can be built using Java, SWIG, CMake and Apple Clang or gcc.

First, install SWIG and Java (also make sure that JAVA_HOME is set properly). Then, either follow the Apple Clang or gcc installation instructions below.

**Apple Clang**

Only Apple Clang version 8.1 or higher is supported.

1. Install CMake (3.16 or higher):

```bash
brew install cmake
```

2. Install OpenMP:

```bash
brew install libomp
```

3. Run the following commands:

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -DUSE_SWIG=ON -DAPPLE_OUTPUT_DYLIB=ON ..
make -j4
```
gcc

1. Install CMake (3.2 or higher):

```
brew install cmake
```

2. Install gcc:

```
brew install gcc
```

3. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=g++-7 CC=gcc-7  # replace "7" with version of gcc installed on your machine
mkdir build
cd build
cmake -DUSE_SWIG=ON -DAPPLE_OUTPUT_DYLIB=ON ..
make -j4
```

Also, you may want to read gcc Tips.
This is a quick start guide for LightGBM CLI version. Follow the Installation Guide to install LightGBM first.

List of other helpful links
- Parameters
- Parameters Tuning
- Python-package Quick Start
- Python API

2.1 Training Data Format

LightGBM supports input data files with CSV, TSV and LibSVM (zero-based) formats.
Files could be both with and without headers.
Label column could be specified both by index and by name.
Some columns could be ignored.

2.1.1 Categorical Feature Support

LightGBM can use categorical features directly (without one-hot encoding). The experiment on Expo data shows about 8x speed-up compared with one-hot encoding.
For the setting details, please refer to the categorical_feature parameter.

2.1.2 Weight and Query/Group Data

LightGBM also supports weighted training, it needs an additional weight data. And it needs an additional query data for ranking task.
Also, weight and query data could be specified as columns in training data in the same manner as label.
2.2 Parameters Quick Look

The parameters format is `key1=value1 key2=value2 ...`. Parameters can be set both in config file and command line. If one parameter appears in both command line and config file, LightGBM will use the parameter from the command line. The most important parameters which new users should take a look to are located into Core Parameters and the top of Learning Control Parameters sections of the full detailed list of LightGBM’s parameters.

2.3 Run LightGBM

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

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

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

2.4 Examples

- Binary Classification
- Regression
- Lambdarank
- Distributed Learning
This document gives a basic walkthrough of LightGBM Python-package.

List of other helpful links
- Python Examples
- Python API
- Parameters Tuning

3.1 Install

The preferred way to install LightGBM is via pip:

```bash
pip install lightgbm
```

Refer to Python-package folder for the detailed installation guide.

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

```python
import lightgbm as lgb
```

3.2 Data Interface

The LightGBM Python module can load data from:

- LibSVM (zero-based) / TSV / CSV / TXT format file
- NumPy 2D array(s), pandas DataFrame, H2O DataTable’s Frame, SciPy sparse matrix
- LightGBM binary file

The data is stored in a `Dataset` object.

Many of the examples in this page use functionality from `numpy`. To run the examples, be sure to import `numpy` in your session.

```python
import numpy as np
```

To load a LibSVM (zero-based) text file or a LightGBM binary file into Dataset:

```python
train_data = lgb.Dataset('train.svm.bin')
```
To load a numpy array into Dataset:

```python
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 scipy.sparse.csr_matrix array into Dataset:

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

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

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

Create validation data:

```python
validation_data = train_data.create_valid('validation.svm')
```

or

```python
validation_data = lgb.Dataset('validation.svm', reference=train_data)
```

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

**Specific feature names and categorical features:**

```python
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 convert to one-hot encoding, and is much faster than one-hot encoding (about 8x speed-up).

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

**Weights can be set when needed:**

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

or

```python
train_data = lgb.Dataset(data, label=label)
w = np.random.randn(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 efficient usage:**

The Dataset object in LightGBM is very memory-efficient, it only needs to save discrete bins. However, Numpy/Array/Pandas object is memory expensive. If you are concerned about your memory consumption, you can save memory by:

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

3.3 Setting Parameters

LightGBM can use a dictionary to set Parameters. For instance:

- **Booster parameters:**
  ```python
  param = {'num_leaves': 31, 'objective': 'binary'}
  param['metric'] = 'auc'
  ```

- You can also specify multiple eval metrics:
  ```python
  param['metric'] = ['auc', 'binary_logloss']
  ```

3.4 Training

Training a model requires a parameter list and data set:

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

After training, the model can be saved:

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

The trained model can also be dumped to JSON format:

```python
json_model = bst.dump_model()
```

A saved model can be loaded:

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

3.5 CV

Training with 5-fold CV:

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

3.6 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 is more than one, it will use all of them except the training data:

```python
bst = lgb.train(param, train_data, num_round, valid_sets=valid_sets, early_stopping_rounds=5)
bst.save_model('model.txt', num_iteration=bst.best_iteration)
```
The model will train until the validation score stops improving. Validation score needs to improve at least every 
early_stopping_rounds to continue training.

The index of iteration that has the best performance will be saved in the best_iteration field if early stopping 
logic is enabled by setting early_stopping_rounds. Note that train() will return a model from the best 
itration.

This works with both metrics to minimize (L2, log loss, etc.) and to maximize (NDCG, AUC, etc.). Note that if you 
specify more than one evaluation metric, all of them will be used for early stopping. However, you can change this be-
havior and make LightGBM check only the first metric for early stopping by passing first_metric_only=True 
in param or early_stopping callback constructor.

3.7 Prediction

A model that has been trained or loaded can perform predictions on datasets:

```python
# 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.

```python
best_iteration:
ypred = bst.predict(data, num_iteration=bst.best_iteration)
```
CHAPTER
FOUR

FEATURES

This is a conceptual overview of how LightGBM works\cite{1}. We assume familiarity with decision tree boosting algo-
rithms to focus instead on aspects of LightGBM that may differ from other boosting packages. For detailed algorithms,
please refer to the citations or source code.

4.1 Optimization in Speed and Memory Usage

Many boosting tools use pre-sort-based algorithms\cite{2, 3} (e.g. default algorithm in xgboost) for decision tree learning.
It is a simple solution, but not easy to optimize.

LightGBM uses histogram-based algorithms\cite{4, 5, 6}, which bucket continuous feature (attribute) values into discrete
bins. This speeds up training and reduces memory usage. Advantages of histogram-based algorithms include the
following:

• **Reduced cost of calculating the gain for each split**
  – Pre-sort-based algorithms have time complexity $O(#\text{data})$
  – Computing the histogram has time complexity $O(#\text{data})$, but this involves only a fast sum-up operation.
    Once the histogram is constructed, a histogram-based algorithm has time complexity $O(#\text{bins})$, and $#\text{bins}$ is far smaller than $#\text{data}$.

• **Use histogram subtraction for further speedup**
  – To get one leaf’s histograms in a binary tree, use the histogram subtraction of its parent and its neighbor
  – So it needs to construct histograms for only one leaf (with smaller $#\text{data}$ than its neighbor). It then can
    get histograms of its neighbor by histogram subtraction with small cost ($O(#\text{bins})$)

• **Reduce memory usage**
  – Replaces continuous values with discrete bins. If $#\text{bins}$ 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 distributed learning**
4.2 Sparse Optimization

• Need only $O(2 \times \#\text{non_zero_data})$ to construct histogram for sparse features

4.3 Optimization in Accuracy

4.3.1 Leaf-wise (Best-first) Tree Growth

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

\[\text{Level-wise tree growth}\]

LightGBM grows trees leaf-wise (best-first)[7]. It will choose the leaf with max delta loss to grow. Holding \#leaf fixed, leaf-wise algorithms tend to achieve lower loss than level-wise algorithms.

Leaf-wise may cause over-fitting when \#data is small, so LightGBM includes the max_depth parameter to limit tree depth. However, trees still grow leaf-wise even when max_depth is specified.

\[\text{Leaf-wise tree growth}\]
4.3.2 Optimal Split for Categorical Features

It is common to represent categorical features with one-hot encoding, but this approach is suboptimal for tree learners. Particularly for high-cardinality categorical features, a tree built on one-hot features tends to be unbalanced and needs to grow very deep to achieve good accuracy.

Instead of one-hot encoding, the optimal solution is to split on a categorical feature by partitioning its categories into 2 subsets. If the feature has \( k \) categories, there are \( 2^{k-1} - 1 \) possible partitions. But there is an efficient solution for regression trees\(^8\). It needs about \( O(k \times \log(k)) \) to find the optimal partition.

The basic idea is to sort the categories according to the training objective at each split. More specifically, LightGBM sorts the histogram (for a categorical feature) according to its accumulated values (\( \text{sum}_{\text{gradient}} / \text{sum}_{\text{hessian}} \)) and then finds the best split on the sorted histogram.

4.4 Optimization in Network Communication

It only needs to use some collective communication algorithms, like “All reduce”, “All gather” and “Reduce scatter”, in distributed learning of LightGBM. LightGBM implements state-of-art algorithms\(^9\). These collective communication algorithms can provide much better performance than point-to-point communication.

4.5 Optimization in Distributed Learning

LightGBM provides the following distributed learning algorithms.

4.5.1 Feature Parallel

Traditional Algorithm

Feature parallel aims to parallelize 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 to received data.

The shortcomings of traditional feature parallel:

- Has computation overhead, since it cannot speed up “split”, whose time complexity is \( O(\#\text{data}) \). Thus, feature parallel cannot speed up well when \( \#\text{data} \) is large.
- Need communication of split result, which costs about \( O(\#\text{data} / 8) \) (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: 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 knows how to split data. And \#data won’t be larger, so it is reasonable to hold the 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.

### 4.5.2 Data Parallel

#### Traditional Algorithm

Data parallel aims to parallelize 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 shortcomings of traditional data parallel:

- High communication cost. If using point-to-point communication algorithm, communication cost for one machine is about $O(\#machine \times \#feature \times \#bin)$. If using collective communication algorithm (e.g. “All Reduce”), communication cost is about $O(2 \times \#feature \times \#bin)$ (check cost of “All Reduce” in chapter 4.5 at [9]).

#### Data Parallel in LightGBM

We reduce communication cost of data parallel in LightGBM:

1. Instead of “Merge global histograms from all local histograms”, LightGBM uses “Reduce Scatter” to merge histograms of different (non-overlapping) features for different workers. Then workers find the local best split on local merged histograms and sync up the global best split.
2. As aforementioned, LightGBM uses 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.

All things considered, data parallel in LightGBM has time complexity $O(0.5 \times \#feature \times \#bin)$. 
4.5.3 Voting Parallel

Voting parallel further reduces the communication cost in Data Parallel to constant cost. It uses two-stage voting to reduce the communication cost of feature histograms[10].

4.6 GPU Support

Thanks @huanzhang12 for contributing this feature. Please read [11] to get more details.

- GPU Installation
- GPU Tutorial

4.7 Applications and Metrics

LightGBM supports the following applications:

- regression, the objective function is L2 loss
- binary classification, the objective function is logloss
- multi classification
- cross-entropy, the objective function is logloss and supports training on non-binary labels
- lambdarank, the objective function is lambdarank with NDCG

LightGBM supports the following metrics:

- L1 loss
- L2 loss
- Log loss
- Classification error rate
- AUC
- NDCG
- MAP
- Multi-class log loss
- Multi-class error rate
- AUC-mu (new in v3.0.0)
- Average precision (new in v3.1.0)
- Fair
- Huber
- Poisson
- Quantile
- MAPE
- Kullback-Leibler
- Gamma
4.8 Other Features

- Limit \( \text{max\_depth} \) 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
- Multiple validation data
- Multiple metrics
- Early stopping (both training and prediction)
- Prediction for leaf index

For more details, please refer to Parameters.

4.9 References


5.1 Comparison Experiment

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

5.1.1 History

08 Mar, 2020: update according to the latest master branch (1b97eaf for XGBoost, bcad692 for LightGBM). (xgboost_exact is not updated for it is too slow.)


5.1.2 Data

We used 5 datasets to conduct our comparison experiments. Details of data are listed in the following table:

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

5.1.3 Environment

We ran all experiments on a single Linux server (Azure ND24s) with the following specifications:

<table>
<thead>
<tr>
<th>OS</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu 16.04 LTS</td>
<td>2 * E5-2690 v4</td>
<td>448GB</td>
</tr>
</tbody>
</table>
5.1.4 Baseline

We used xgboost as a baseline.
Both xgboost and LightGBM were built with OpenMP support.

5.1.5 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
   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 trees depth-wise and controls model complexity by max_depth. LightGBM uses a 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_leaves=255.

Other parameters are default values.

5.1.6 Result

Speed

We compared speed using only the training task without any test or metric output. We didn’t count the time for IO. For the ranking tasks, since XGBoost and LightGBM implement different ranking objective functions, we used regression objective for speed benchmark, for the fair comparison.

The following table is the comparison of time cost:
LightGBM ran faster than xgboost on all experiment data sets.

Accuracy

We computed all accuracy metrics only on the test data set.

<table>
<thead>
<tr>
<th>Data</th>
<th>Metric</th>
<th>xgboost</th>
<th>xgboost_hist</th>
<th>LightGBM</th>
<th>LightGBM (col-wise)</th>
<th>LightGBM (row-wise)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higgs</td>
<td>AUC</td>
<td>0.839593</td>
<td>0.845314</td>
<td>0.845724</td>
<td>0.897GB</td>
<td>1.401GB</td>
</tr>
<tr>
<td>Yahoo LTR</td>
<td>NDCG1</td>
<td>0.719748</td>
<td>0.720049</td>
<td>0.732981</td>
<td>1.907GB</td>
<td>4.023GB</td>
</tr>
<tr>
<td></td>
<td>NDCG3</td>
<td>0.717813</td>
<td>0.722573</td>
<td>0.735689</td>
<td>1.741GB</td>
<td>2.161GB</td>
</tr>
<tr>
<td></td>
<td>NDCG5</td>
<td>0.737849</td>
<td>0.740899</td>
<td>0.75352</td>
<td>0.940GB</td>
<td>1.296GB</td>
</tr>
<tr>
<td></td>
<td>NDCG10</td>
<td>0.78089</td>
<td>0.782957</td>
<td>0.793498</td>
<td>0.492429</td>
<td>0.524252</td>
</tr>
<tr>
<td>MS LTR</td>
<td>NDCG1</td>
<td>0.483956</td>
<td>0.485115</td>
<td>0.517767</td>
<td>0.555GB</td>
<td>1.116GB</td>
</tr>
<tr>
<td></td>
<td>NDCG3</td>
<td>0.467951</td>
<td>0.47313</td>
<td>0.501063</td>
<td>0.555GB</td>
<td>1.755GB</td>
</tr>
<tr>
<td></td>
<td>NDCG5</td>
<td>0.472476</td>
<td>0.476375</td>
<td>0.504648</td>
<td>0.555GB</td>
<td>1.755GB</td>
</tr>
<tr>
<td></td>
<td>NDCG10</td>
<td>0.492429</td>
<td>0.496553</td>
<td>0.524252</td>
<td>0.555GB</td>
<td>1.755GB</td>
</tr>
<tr>
<td>Expo</td>
<td>AUC</td>
<td>0.756713</td>
<td>0.776224</td>
<td>0.776935</td>
<td>0.607201</td>
<td>0.609465</td>
</tr>
<tr>
<td>Allstate</td>
<td>AUC</td>
<td>0.607201</td>
<td>0.609465</td>
<td>0.609072</td>
<td>0.607201</td>
<td>0.609465</td>
</tr>
</tbody>
</table>

Memory Consumption

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

<table>
<thead>
<tr>
<th>Data</th>
<th>xgboost</th>
<th>xgboost_hist</th>
<th>LightGBM (col-wise)</th>
<th>LightGBM (row-wise)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higgs</td>
<td>4.853GB</td>
<td>7.335GB</td>
<td>0.897GB</td>
<td>1.401GB</td>
</tr>
<tr>
<td>Yahoo LTR</td>
<td>1.907GB</td>
<td>4.023GB</td>
<td>1.741GB</td>
<td>2.161GB</td>
</tr>
<tr>
<td>MS LTR</td>
<td>5.469GB</td>
<td>7.491GB</td>
<td>0.940GB</td>
<td>1.296GB</td>
</tr>
<tr>
<td>Expo</td>
<td>1.553GB</td>
<td>2.606GB</td>
<td>0.555GB</td>
<td>0.711GB</td>
</tr>
<tr>
<td>Allstate</td>
<td>6.237GB</td>
<td>12.090GB</td>
<td>1.116GB</td>
<td>1.755GB</td>
</tr>
</tbody>
</table>

5.2 Parallel Experiment

5.2.1 History

5.2.2 Data

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

<table>
<thead>
<tr>
<th>Data</th>
<th>Task</th>
<th>Link</th>
<th>#Data</th>
<th>#Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Criteo</td>
<td>Binary classification</td>
<td>link</td>
<td>1,700,000,000</td>
<td>67</td>
</tr>
</tbody>
</table>

This data contains 13 integer features and 26 categorical features for 24 days of click logs. We statisticized the clickthrough rate (CTR) and count for these 26 categorical features from the first ten days. Then we used the next ten days’ data, after replacing the categorical features by the corresponding CTR and count, as training data. The processed training data have a total of 1.7 billions records and 67 features.

5.2.3 Environment

We ran our experiments on 16 Windows servers with the following specifications:

<table>
<thead>
<tr>
<th>OS</th>
<th>CPU</th>
<th>Memory</th>
<th>Network Adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 2012 Server</td>
<td>E5-2670 v2 DDR3 1600Mhz, 256GB</td>
<td>Mellanox ConnectX-3, 54Gbps, RDMA support</td>
<td></td>
</tr>
</tbody>
</table>

5.2.4 Settings

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

We used data parallel here because this data is large in #data but small in #feature. Other parameters were default values.

5.2.5 Results

<table>
<thead>
<tr>
<th>#Machine</th>
<th>Time per Tree</th>
<th>Memory Usage(per Machine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>627.8 s</td>
<td>176GB</td>
</tr>
<tr>
<td>2</td>
<td>311 s</td>
<td>87GB</td>
</tr>
<tr>
<td>4</td>
<td>156 s</td>
<td>43GB</td>
</tr>
<tr>
<td>8</td>
<td>80 s</td>
<td>22GB</td>
</tr>
<tr>
<td>16</td>
<td>42 s</td>
<td>11GB</td>
</tr>
</tbody>
</table>

The results show that LightGBM achieves a linear speedup with distributed learning.
5.3 GPU Experiments

Refer to GPU Performance.
This page contains descriptions of all parameters in LightGBM.

List of other helpful links

- Python API
- Parameters Tuning

External Links

- Laurae++ Interactive Documentation

6.1 Parameters Format

The parameters format is key1=value1 key2=value2 .... 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 from the command line.

6.2 Core Parameters

- config, default = " ", type = string, aliases: config_file
  - path of config file
  - Note: can be used only in CLI version
- task, default = train, type = enum, options: train, predict, convert_model, refit, aliases: task_type
  - train, for training, aliases: training
  - predict, for prediction, aliases: prediction, test
  - convert_model, for converting model file into if-else format, see more information in Convert Parameters
  - refit, for refitting existing models with new data, aliases: refit_tree
- save_binary, load train (and validation) data then save dataset to binary file. Typical usage: save_binary first, then run multiple train tasks in parallel using the saved binary file
- **Note**: can be used only in CLI version; for language-specific packages you can use the correspondent functions

- **objective**, default = **regression**, type = **enum**, options: regression, regression_l1, huber, fair, poisson, quantile, mape, gamma, tweedie, binary, multiclass, multiclassova, cross_entropy, cross_entropy_lambda, lambdarank, rank_xendcg, aliases: objective_type, app, application

  - **regression application**
    - regression, L2 loss, aliases: regression_l2, l2, mean_squared_error, mse, l2_root, root_mean_squared_error, rmse
    - regression_l1, L1 loss, aliases: l1, mean_absolute_error, mae
    - huber, Huber loss
    - fair, Fair loss
    - poisson, Poisson regression
    - quantile, Quantile regression
    - mape, MAPE loss, aliases: mean_absolute_percentage_error
    - gamma, Gamma regression with log-link. It might be useful, e.g., for modeling insurance claims severity, or for any target that might be gamma-distributed
    - tweedie, Tweedie regression with log-link. It might be useful, e.g., for modeling total loss in insurance, or for any target that might be tweedie-distributed

  - **binary classification application**
    - binary, binary log loss classification (or logistic regression)
    - requires labels in \( \{0, 1\} \); see cross-entropy application for general probability labels in \([0, 1]\)

  - **multi-class classification application**
    - multiclass, softmax objective function, aliases: softmax
    - multiclassova, One-vs-All binary objective function, aliases: multiclass_ova, ova, ovr
    - num_class should be set as well

  - **cross-entropy application**
    - cross_entropy, objective function for cross-entropy (with optional linear weights), aliases: xentropy
    - cross_entropy_lambda, alternative parameterization of cross-entropy, aliases: xentlambda
    - label is anything in interval \([0, 1]\)

  - **ranking application**
    - lambdarank, lambdarank objective. label_gain can be used to set the gain (weight) of int label and all values in label must be smaller than number of elements in label_gain
    - rank_xendcg, XE_NDCG_MART ranking objective function, aliases: xendcg, xe_ndcg, xe_ndcg_mart, xendcg_mart
    - rank_xendcg is faster than and achieves the similar performance as lambdarank
    - label should be int type, and larger number represents the higher relevance (e.g. 0:bad, 1:fair, 2:good, 3:perfect)
• **boosting**, default = **gbdt**, type = **enum**, options: gbdt, rf, dart, goss, aliases: boosting_type, boost
  - gbdt, traditional Gradient Boosting Decision Tree, aliases: gbrt
  - rf, Random Forest, aliases: random_forest
  - dart, Dropouts meet Multiple Additive Regression Trees
  - goss, Gradient-based One-Side Sampling
    * **Note**: internally, LightGBM uses gbdt mode for the first \( \frac{1}{\text{learning_rate}} \) iterations
• **linear_tree**, default = false, type = **bool**
  - fit piecewise linear gradient boosting tree
    * tree splits are chosen in the usual way, but the model at each leaf is linear instead of constant
    * the linear model at each leaf includes all the numerical features in that leaf’s branch
    * categorical features are used for splits as normal but are not used in the linear models
    * missing values should not be encoded as 0. Use np.nan for Python, NA for the CLI, and NA, NA_real_, or NA_integer_ for R
    * it is recommended to rescale data before training so that features have similar mean and standard deviation
    * **Note**: only works with CPU and serial tree learner
    * **Note**: regression_l1 objective is not supported with linear tree boosting
    * **Note**: setting linear_tree=true significantly increases the memory use of LightGBM
    * **Note**: if you specify monotone_constraints, constraints will be enforced when choosing the split points, but not when fitting the linear models on leaves
• **data**, default = "", type = **string**, aliases: train, train_data, train_data_file, data_filename
  - path of training data, LightGBM will train from this data
  - **Note**: can be used only in CLI version
• **valid**, default = "", type = **string**, aliases: test, valid_data, valid_data_file, test_data, test_data_file, valid_filenames
  - path(s) of validation/test data, LightGBM will output metrics for these data
  - support multiple validation data, separated by ,
  - **Note**: can be used only in CLI version
• **num_iterations**, default = 100, type = **int**, aliases: num_iteration, n_iter, num_tree, num_trees, num_round, num_rounds, num_boost_round, n_estimators, constraints: num_iterations \( \geq 0 \)
  - number of boosting iterations
  - **Note**: internally, LightGBM constructs \( \text{num_class} \times \text{num_iterations} \) trees for multi-class classification problems
• **learning_rate**, default = 0.1, type = **double**, aliases: shrinkage_rate, eta, constraints: learning_rate \( > 0.0 \)
  - shrinkage rate
  - in dart, it also affects on normalization weights of dropped trees
• **num_leaves**, default = 31, type = int, aliases: num_leaf, max_leaves, max_leaf, constraints: 1 < num_leaves <= 131072
  - max number of leaves in one tree

• **tree_learner**, default = serial, type = enum, options: serial, feature, data, voting, aliases: tree, tree_type, tree_learner_type
  - serial, single machine tree learner
  - feature, feature parallel tree learner, aliases: feature_parallel
  - data, data parallel tree learner, aliases: data_parallel
  - voting, voting parallel tree learner, aliases: voting_parallel
  - refer to [Distributed Learning Guide](#) to get more details

• **num_threads**, default = 0, type = int, aliases: num_thread, nthread, nthreads, n_jobs
  - number of threads for LightGBM
  - 0 means default number of threads in OpenMP
  - for the best speed, set this to the number of **real CPU cores**, not the number of threads (most CPUs use hyper-threading to generate 2 threads per CPU core)
  - do not set it too large if your dataset is small (for instance, do not use 64 threads for a dataset with 10,000 rows)
  - be aware a task manager or any similar CPU monitoring tool might report that cores not being fully utilized. **This is normal**
  - for distributed learning, do not use all CPU cores because this will cause poor performance for the network communication
  - **Note**: please don’t change this during training, especially when running multiple jobs simultaneously by external packages, otherwise it may cause undesirable errors

• **device_type**, default = cpu, type = enum, options: cpu, gpu, cuda, aliases: device
  - device for the tree learning, you can use GPU to achieve the faster learning
  - **Note**: it is recommended to use the smaller max_bin (e.g. 63) to get the better speed up
  - **Note**: for the faster speed, GPU uses 32-bit float point to sum up by default, so this 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
  - **Note**: refer to [Installation Guide](#) to build LightGBM with GPU support

• **seed**, default = None, type = int, aliases: random_seed, random_state
  - this seed is used to generate other seeds, e.g. data_random_seed, feature_fraction_seed, etc.
  - by default, this seed is unused in favor of default values of other seeds
  - this seed has lower priority in comparison with other seeds, which means that it will be overridden, if you set other seeds explicitly

• **deterministic**, default = false, type = bool
  - used only with cpu device type
  - setting this to true should ensure the stable results when using the same data and the same parameters (and different num_threads)
– when you use the different seeds, different LightGBM versions, the binaries compiled by different compilers, or in different systems, the results are expected to be different
– you can raise issues in LightGBM GitHub repo when you meet the unstable results
– **Note:** setting this to `true` may slow down the training
– **Note:** to avoid potential instability due to numerical issues, please set `force_col_wise=true` or `force_row_wise=true` when setting `deterministic=true`

### 6.3 Learning Control Parameters

- **force_col_wise**, default = `false`, type = bool
  - used only with `cpu` device type
  - set this to `true` to force col-wise histogram building
  - enabling this is recommended when:
    * the number of columns is large, or the total number of bins is large
    * `num_threads` is large, e.g. > 20
    * you want to reduce memory cost
  - **Note:** when both `force_col_wise` and `force_row_wise` are `false`, LightGBM will firstly try them both, and then use the faster one. To remove the overhead of testing set the faster one to `true` manually
  - **Note:** this parameter cannot be used at the same time with `force_row_wise`, choose only one of them

- **force_row_wise**, default = `false`, type = bool
  - used only with `cpu` device type
  - set this to `true` to force row-wise histogram building
  - enabling this is recommended when:
    * the number of data points is large, and the total number of bins is relatively small
    * `num_threads` is relatively small, e.g. <= 16
    * you want to use small `bagging_fraction` or `goss` boosting to speed up
  - **Note:** setting this to `true` will double the memory cost for Dataset object. If you have not enough memory, you can try setting `force_col_wise=true`
  - **Note:** when both `force_col_wise` and `force_row_wise` are `false`, LightGBM will firstly try them both, and then use the faster one. To remove the overhead of testing set the faster one to `true` manually
  - **Note:** this parameter cannot be used at the same time with `force_col_wise`, choose only one of them

- **histogram_pool_size**, default = `-1.0`, type = double, aliases: `hist_pool_size`
  - max cache size in MB for historical histogram
  - `< 0` means no limit

- **max_depth**, default = `-1`, type = int
  - limit the max depth for tree model. This is used to deal with over-fitting when `#data` is small. Tree still grows leaf-wise
- \( \leq 0 \) means no limit

- **min_data_in_leaf**, default = 20, type = int, aliases: min_data_per_leaf, min_data, min_child_samples, constraints: \( \text{min_data_in_leaf} \geq 0 \)
  - minimal number of data in one leaf. Can be used to deal with over-fitting
  - **Note**: this is an approximation based on the Hessian, so occasionally you may observe splits which produce leaf nodes that have less than this many observations

- **min_sum_hessian_in_leaf**, default = \( 1e-3 \), type = double, aliases: min_sum_hessian_per_leaf, min_hessian, min_child_weight, constraints: \( \text{min_sum_hessian_in_leaf} \geq 0.0 \)
  - minimal sum hessian in one leaf. Like `min_data_in_leaf`, it can be used to deal with over-fitting

- **bagging_fraction**, default = 1.0, type = double, aliases: sub_row, subsample, bagging, constraints: \( 0.0 < \text{bagging_fraction} \leq 1.0 \)
  - like `feature_fraction`, but this will randomly select part of data without resampling
  - can be used to speed up training
  - can be used to deal with over-fitting
  - **Note**: to enable bagging, `bagging_freq` should be set to a non zero value as well

- **pos_bagging_fraction**, default = 1.0, type = double, aliases: pos_sub_row, pos_subsample, pos_bagging, constraints: \( 0.0 < \text{pos_bagging_fraction} \leq 1.0 \)
  - used only in binary application
  - used for imbalanced binary classification problem, will randomly sample \( \#\text{pos_samples} \times \text{pos_bagging_fraction} \) positive samples in bagging
  - should be used together with `neg_bagging_fraction`
  - set this to 1.0 to disable
  - **Note**: to enable this, you need to set `bagging_freq` and `neg_bagging_fraction` as well
  - **Note**: if both `pos_bagging_fraction` and `neg_bagging_fraction` are set to 1.0, balanced bagging is disabled
  - **Note**: if balanced bagging is enabled, `bagging_fraction` will be ignored

- **neg_bagging_fraction**, default = 1.0, type = double, aliases: neg_sub_row, neg_subsample, neg_bagging, constraints: \( 0.0 < \text{neg_bagging_fraction} \leq 1.0 \)
  - used only in binary application
  - used for imbalanced binary classification problem, will randomly sample \( \#\text{neg_samples} \times \text{neg_bagging_fraction} \) negative samples in bagging
  - should be used together with `pos_bagging_fraction`
  - set this to 1.0 to disable
  - **Note**: to enable this, you need to set `bagging_freq` and `pos_bagging_fraction` as well
  - **Note**: if both `pos_bagging_fraction` and `neg_bagging_fraction` are set to 1.0, balanced bagging is disabled
  - **Note**: if balanced bagging is enabled, `bagging_fraction` will be ignored

- **bagging_freq**, default = 0, type = int, aliases: subsample_freq
  - frequency for bagging
0 means disable bagging; k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations.

Note: to enable bagging, bagging_fraction should be set to value smaller than 1.0 as well.

- **bagging_seed**, default = 3, type = int, aliases: bagging_fraction_seed
  - random seed for bagging

- **feature_fraction**, default = 1.0, type = double, aliases: sub_feature, colsample_bytree, constraints: 0.0 < feature_fraction <= 1.0
  - LightGBM will randomly select a subset of features on each iteration (tree) if feature_fraction is smaller than 1.0. For example, if you set it to 0.8, LightGBM will select 80% of features before training each tree.
  - can be used to speed up training
  - can be used to deal with over-fitting

- **feature_fraction_bynode**, default = 1.0, type = double, aliases: sub_feature_bynode, colsample_bynode, constraints: 0.0 < feature_fraction_bynode <= 1.0
  - LightGBM will randomly select a subset of features on each tree node if feature_fraction_bynode is smaller than 1.0. For example, if you set it to 0.8, LightGBM will select 80% of features at each tree node.
  - can be used to deal with over-fitting
  - Note: unlike feature_fraction, this cannot speed up training
  - Note: if both feature_fraction and feature_fraction_bynode are smaller than 1.0, the final fraction of each node is feature_fraction * feature_fraction_bynode

- **feature_fraction_seed**, default = 2, type = int
  - random seed for feature_fraction

- **extra_trees**, default = false, type = bool
  - use extremely randomized trees
  - if set to true, when evaluating node splits LightGBM will check only one randomly-chosen threshold for each feature
  - can be used to speed up training
  - can be used to deal with over-fitting

- **extra_seed**, default = 6, type = int
  - random seed for selecting thresholds when extra_trees is true

- **early_stopping_round**, default = 0, type = int, aliases: early_stopping_rounds, early_stopping, n_iter_no_change
  - will stop training if one metric of one validation data doesn’t improve in last early_stopping_round rounds
  - <= 0 means disable
  - can be used to speed up training

- **first_metric_only**, default = false, type = bool
  - LightGBM allows you to provide multiple evaluation metrics. Set this to true, if you want to use only the first metric for early stopping
• **max_delta_step**, default = 0.0, type = double, aliases: max_tree_output, max_leaf_output
  - used to limit the max output of tree leaves
  - <= 0 means no constraint
  - the final max output of leaves is learning_rate * max_delta_step
• **lambda_l1**, default = 0.0, type = double, aliases: reg_alpha, constraints: lambda_l1 >= 0.0
  - L1 regularization
• **lambda_l2**, default = 0.0, type = double, aliases: reg_lambda, lambda, constraints: lambda_l2 >= 0.0
  - L2 regularization
• **linear_lambda**, default = 0.0, type = double, constraints: linear_lambda >= 0.0
  - linear tree regularization, corresponds to the parameter lambda in Eq. 3 of Gradient Boosting with Piece-Wise Linear Regression Trees
• **min_gain_to_split**, default = 0.0, type = double, aliases: min_split_gain, constraints: min_gain_to_split >= 0.0
  - the minimal gain to perform split
  - can be used to speed up training
• **drop_rate**, default = 0.1, type = double, aliases: rate_drop, constraints: 0.0 <= drop_rate <= 1.0
  - used only in dart
  - dropout rate: a fraction of previous trees to drop during the dropout
• **max_drop**, default = 50, type = int
  - used only in dart
  - max number of dropped trees during one boosting iteration
  - <=0 means no limit
• **skip_drop**, default = 0.5, type = double, constraints: 0.0 <= skip_drop <= 1.0
  - used only in dart
  - probability of skipping the dropout procedure during a boosting iteration
• **xgboost_dart_mode**, default = false, type = bool
  - used only in dart
  - set this to true, if you want to use xgboost dart mode
• **uniform_drop**, default = false, type = bool
  - used only in dart
  - set this to true, if you want to use uniform drop
• **drop_seed**, default = 4, type = int
  - used only in dart
  - random seed to choose dropping models
• **top_rate**, default = 0.2, type = double, constraints: 0.0 <= top_rate <= 1.0
• **other_rate**, default = 0.1, type = double, constraints: 0.0 <= other_rate <= 1.0
  – used only in goss
  – the retain ratio of large gradient data

• **min_data_per_group**, default = 100, type = int, constraints: min_data_per_group > 0
  – minimal number of data per categorical group

• **max_cat_threshold**, default = 32, type = int, constraints: max_cat_threshold > 0
  – used for the categorical features
  – limit number of split points considered for categorical features. See the documentation on how LightGBM finds optimal splits for categorical features for more details
  – can be used to speed up training

• **cat_l2**, default = 10.0, type = double, constraints: cat_l2 >= 0.0
  – used for the categorical features
  – L2 regularization in categorical split

• **cat_smooth**, default = 10.0, type = double, constraints: cat_smooth >= 0.0
  – used for the categorical features
  – this can reduce the effect of noises in categorical features, especially for categories with few data

• **max_cat_to_onehot**, default = 4, type = int, constraints: max_cat_to_onehot > 0
  – when number of categories of one feature smaller than or equal to max_cat_to_onehot, one-vs-other split algorithm will be used

• **top_k**, default = 20, type = int, aliases: topk, constraints: top_k > 0
  – used only in voting tree learner, refer to Voting parallel
  – set this to larger value for more accurate result, but it will slow down the training speed

• **monotone_constraints**, default = None, type = multi-int, aliases: mc, monotone_constraint
  – used for constraints of monotonic features
  – 1 means increasing, -1 means decreasing, 0 means non-constraint
  – you need to specify all features in order. For example, mc=-1,0,1 means decreasing for 1st feature, non-constraint for 2nd feature and increasing for the 3rd feature

• **monotone_constraints_method**, default = basic, type = enum, options: basic, intermediate, advanced, aliases: monotone_constraining_method, mc_method
  – used only if monotone_constraints is set
  – monotone constraints method
    * basic, the most basic monotone constraints method. It does not slow the library at all, but over-constrains the predictions
    * intermediate, a more advanced method, which may slow the library very slightly. However, this method is much less constraining than the basic method and should significantly improve the results

6.3. Learning Control Parameters 45
* advanced, an even more advanced method, which may slow the library. However, this method is even less constraining than the intermediate method and should again significantly improve the results

- **monotone_penalty**, default = 0.0, type = double, aliases: monotone_splits_penalty, ms_penalty, mc_penalty, constraints: monotone_penalty \( \geq 0.0 \)
  - used only if monotone_constraints is set
  - monotone penalty: a penalization parameter \( X \) forbids any monotone splits on the first \( X \) (rounded down) level(s) of the tree. The penalty applied to monotone splits on a given depth is a continuous, increasing function the penalization parameter
  - if 0.0 (the default), no penalization is applied

- **feature_contri**, default = None, type = multi-double, aliases: feature_contrib, fc, fp, feature_penalty
  - used to control feature’s split gain, will use \( \text{gain}[i] = \text{max}(0, \text{feature_contri}[i]) \times \text{gain}[i] \) to replace the split gain of \( i \)-th feature
  - you need to specify all features in order

- **forcedsplits_filename**, default = "", type = string, aliases: fs, forced_splits_filename, forced_splits_file, forced_splits
  - path to a .json file that specifies splits to force at the top of every decision tree before best-first learning commences
  - .json file can be arbitrarily nested, and each split contains feature, threshold fields, as well as left and right fields representing subsplits
  - categorical splits are forced in a one-hot fashion, with left representing the split containing the feature value and right representing other values
  - **Note**: the forced split logic will be ignored, if the split makes gain worse
  - see this file as an example

- **refit_decay_rate**, default = 0.9, type = double, constraints: 0.0 \( \leq \) refit_decay_rate \( \leq \) 1.0
  - decay rate of refit task, will use \( \text{leaf_output} = \text{refit_decay_rate} \times \text{old_leaf_output} + (1.0 - \text{refit_decay_rate}) \times \text{new_leaf_output} \) to refit trees
  - used only in refit task in CLI version or as argument in refit function in language-specific package

- **cegb_tradeoff**, default = 1.0, type = double, constraints: cegb_tradeoff \( \geq \) 0.0
  - cost-effective gradient boosting multiplier for all penalties

- **cegb_penalty_split**, default = 0.0, type = double, constraints: cegb_penalty_split \( \geq \) 0.0
  - cost-effective gradient-boosting penalty for splitting a node

- **cegb_penalty_feature_lazy**, default = 0, 0, ..., 0, type = multi-double
  - cost-effective gradient boosting penalty for using a feature
  - applied per data point

- **cegb_penalty_feature_coupled**, default = 0, 0, ..., 0, type = multi-double
  - cost-effective gradient boosting penalty for using a feature
  - applied once per forest

- **path_smooth**, default = 0, type = double, constraints: path_smooth \( \geq \) 0.0
controls smoothing applied to tree nodes
- helps prevent overfitting on leaves with few samples
- if set to zero, no smoothing is applied
- if \( \text{path_smooth} > 0 \) then \( \text{min_data_in_leaf} \) must be at least 2
- larger values give stronger regularization
  * the weight of each node is \((n / \text{path_smooth}) \times w + w_p / (n / \text{path_smooth} + 1)\),
    where \( n \) is the number of samples in the node, \( w \) is the optimal node weight to minimise the
    loss (approximately \(-\text{sum_gradients} / \text{sum_hessians}\)) , and \( w_p \) is the weight of the parent
    node
  * note that the parent output \( w_p \) itself has smoothing applied, unless it is the root node, so that the
    smoothing effect accumulates with the tree depth

• interaction_constraints, default = "", type = string
  - controls which features can appear in the same branch
  - by default interaction constraints are disabled, to enable them you can specify
    * for CLI, lists separated by commas, e.g. \([0, 1, 2], [2, 3]\)
    * for Python-package, list of lists, e.g. \([[0, 1, 2]], [2, 3]]\)
    * for R-package, list of character or numeric vectors, e.g. \(\text{list}(c("var1", "var2", "var3"), c("var3", "var4"))\) or \(\text{list}(\text{c(1L, 2L, 3L), c(3L, 4L)})\). Numeric
      vectors should use 1-based indexing, where \(1L\) is the first feature, \(2L\) is the second feature, etc
  - any two features can only appear in the same branch only if there exists a constraint containing both
    features

• verbosity, default = 1, type = int, aliases: verbose
  - controls the level of LightGBM’s verbosity
  - \( < 0 \): Fatal, \( = 0 \): Error (Warning), \( = 1 \): Info, \( > 1 \): Debug

• input_model, default = "", type = string, aliases: model_input, model_in
  - filename of input model
  - for prediction task, this model will be applied to prediction data
  - for train task, training will be continued from this model
  - Note: can be used only in CLI version

• output_model, default = LightGBM_model.txt, type = string, aliases: model_output, model_out
  - filename of output model in training
  - Note: can be used only in CLI version

• saved_feature_importance_type, default = 0, type = int
  - the feature importance type in the saved model file
  - \( 0 \): count-based feature importance (numbers of splits are counted); \( 1 \): gain-based feature importance
    (values of gain are counted)
  - Note: can be used only in CLI version

• snapshot_freq, default = -1, type = int, aliases: save_period

6.3. Learning Control Parameters 47
– frequency of saving model file snapshot
– set this to positive value to enable this function. For example, the model file will be snapshotted at each iteration if snapshot_freq=1
– Note: can be used only in CLI version

6.4 IO Parameters

6.4.1 Dataset Parameters

• max_bin, default = 255, type = int, constraints: max_bin > 1
  – max number of bins that feature values will be bucketed in
  – small number of bins may reduce training accuracy but may increase general power (deal with over-fitting)
  – LightGBM will auto compress memory according to max_bin. For example, LightGBM will use uint8_t for feature value if max_bin=255
• max_bin_by_feature, default = None, type = multi-int
  – max number of bins for each feature
  – if not specified, will use max_bin for all features
• min_data_in_bin, default = 3, type = int, constraints: min_data_in_bin > 0
  – minimal number of data inside one bin
  – use this to avoid one-data-one-bin (potential over-fitting)
• bin_construct_sample_cnt, default = 200000, type = int, aliases: subsample_for_bin, constraints: bin_construct_sample_cnt > 0
  – number of data that sampled to construct feature discrete bins
  – setting this to larger value will give better training result, but may increase data loading time
  – set this to larger value if data is very sparse
  – Note: don’t set this to small values, otherwise, you may encounter unexpected errors and poor accuracy
• data_random_seed, default = 1, type = int, aliases: data_seed
  – random seed for sampling data to construct histogram bins
• is_enable_sparse, default = true, type = bool, aliases: is_sparse, enable_sparse, sparse
  – used to enable/disable sparse optimization
• enable_bundle, default = true, type = bool, aliases: is_enable_bundle, bundle
  – set this to false to disable Exclusive Feature Bundling (EFB), which is described in LightGBM: A Highly Efficient Gradient Boosting Decision Tree
  – Note: disabling this may cause the slow training speed for sparse datasets
• use_missing, default = true, type = bool
  – set this to false to disable the special handle of missing value
• zero_as_missing, default = false, type = bool
– set this to true to treat all zero as missing values (including the unshown values in LibSVM / sparse matrices)
– set this to false to use na for representing missing values

• **feature_pre_filter**, default = true, type = bool
– set this to true (the default) to tell LightGBM to ignore the features that are unsplittable based on min_data_in_leaf
– as dataset object is initialized only once and cannot be changed after that, you may need to set this to false when searching parameters with min_data_in_leaf, otherwise features are filtered by min_data_in_leaf firstly if you don’t reconstruct dataset object
– **Note:** setting this to false may slow down the training

• **pre_partition**, default = false, type = bool, aliases: is_pre_partition
– used for distributed learning (excluding the feature_parallel mode)
– true if training data are pre-partitioned, and different machines use different partitions

• **two_round**, default = false, type = bool, aliases: two_round_loading, use_two_round_loading
– set this to true if data file is too big to fit in memory
– by default, LightGBM will map data file to memory and load features from memory. This will provide faster data loading speed, but may cause run out of memory error when the data file is very big
– **Note:** works only in case of loading data directly from file

• **header**, default = false, type = bool, aliases: has_header
– set this to true if input data has header
– **Note:** works only in case of loading data directly from file

• **label_column**, default = "", type = int or string, aliases: label
– used to specify 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
– **Note:** works only in case of loading data directly from file

• **weight_column**, default = "", type = int or string, aliases: weight
– used to specify 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:** works only in case of loading data directly from file
– **Note:** index starts from 0 and it doesn’t count the label column when passing type is int, e.g. when label is column_0, and weight is column_1, the correct parameter is weight=0

• **group_column**, default = "", type = int or string, aliases: group, group_id, query_column, query, query_id
– used to specify 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:** works only in case of loading data directly from file
– **Note:** data should be grouped by query_id, for more information, see *Query Data*
– **Note:** index starts from 0 and it doesn’t count the label column when passing type is `int`, e.g. when label is column_0 and query_id is column_1, the correct parameter is query=0

- **ignore_column**, default = "", type = multi-int or string, aliases: ignore_feature, blacklist
  - used to specify some ignoring 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:** works only in case of loading data directly from file
  - **Note:** index starts from 0 and it doesn’t count the label column when passing type is `int`
  - **Note:** despite the fact that specified columns will be completely ignored during the training, they still should have a valid format allowing LightGBM to load file successfully

- **categorical_feature**, default = "", type = multi-int or string, aliases: cat_feature, categorical_column, cat_column
  - used to specify 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 supports categorical with `int` type (not applicable for data represented as pandas DataFrame in Python-package)
  - **Note:** index starts from 0 and it doesn’t count the label column when passing type is `int`
  - **Note:** all values should be less than `Int32.MaxValue` (2147483647)
  - **Note:** using large values could be memory consuming. Tree decision rule works best when categorical features are presented by consecutive integers starting from zero
  - **Note:** all negative values will be treated as *missing values*
  - **Note:** the output cannot be monotonically constrained with respect to a categorical feature

- **forcedbins_filename**, default = "", type = string
  - path to a `.json` file that specifies bin upper bounds for some or all features
  - `.json` file should contain an array of objects, each containing the word `feature` (integer feature index) and `bin_upper_bound` (array of thresholds for binning)
  - see this file as an example

- **save_binary**, default = false, type = bool, aliases: is_save_binary, is_save_binary_file
  - if true, LightGBM will save the dataset (including validation data) to a binary file. This speed ups the data loading for the next time
  - **Note:** `init_score` is not saved in binary file
  - **Note:** can be used only in CLI version; for language-specific packages you can use the correspondent function
6.4.2 Predict Parameters

- **start_iteration_predict**, default = 0, type = int
  - used only in prediction task
  - used to specify from which iteration to start the prediction
  - <= 0 means from the first iteration
- **num_iteration_predict**, default = -1, type = int
  - used only in prediction task
  - used to specify how many trained iterations will be used in prediction
  - <= 0 means no limit
- **predict_raw_score**, default = false, type = bool, aliases: is_predict_raw_score, predict_rawscore, raw_score
  - used only in prediction task
  - set this to true to predict only the raw scores
  - set this to false to predict transformed scores
- **predict_leaf_index**, default = false, type = bool, aliases: is_predict_leaf_index, leaf_index
  - used only in prediction task
  - set this to true to predict with leaf index of all trees
- **predict_contrib**, default = false, type = bool, aliases: is_predict_contrib, contrib
  - used only in prediction task
  - set this to true to estimate SHAP values, which represent how each feature contributes to each prediction
  - produces $\#\text{features} + 1$ values where the last value is the expected value of the model output over the training data
  - **Note**: if you want to get more explanation for your model’s predictions using SHAP values like SHAP interaction values, you can install shap package
  - **Note**: unlike the shap package, with predict_contrib we return a matrix with an extra column, where the last column is the expected value
  - **Note**: this feature is not implemented for linear trees
- **predict_disable_shape_check**, default = false, type = bool
  - used only in prediction task
  - control whether or not LightGBM raises an error when you try to predict on data with a different number of features than the training data
  - if false (the default), a fatal error will be raised if the number of features in the dataset you predict on differs from the number seen during training
  - if true, LightGBM will attempt to predict on whatever data you provide. This is dangerous because you might get incorrect predictions, but you could use it in situations where it is difficult or expensive to generate some features and you are very confident that they were never chosen for splits in the model
  - **Note**: be very careful setting this parameter to true
- **pred_early_stop**, default = false, type = bool
– used only in prediction task
– if true, will use early-stopping to speed up the prediction. May affect the accuracy

- pred_early_stop_freq, default = 10, type = int
– used only in prediction task
– the frequency of checking early-stopping prediction

- pred_early_stop_margin, default = 10.0, type = double
– used only in prediction task
– the threshold of margin in early-stopping prediction

- output_result, default = LightGBM_predict_result.txt, type = string, aliases: predict_result, prediction_result, predict_name, prediction_name, pred_name, name_pred
– used only in prediction task
– filename of prediction result
– Note: can be used only in CLI version

### 6.4.3 Convert Parameters

- convert_model_language, default = "", type = string
– used only in convert_model task
– only cpp is supported yet; for conversion model to other languages consider using m2cgen utility
– if convert_model_language is set and task=train, the model will be also converted
– Note: can be used only in CLI version

- convert_model, default = gbdt_prediction.cpp, type = string, aliases: convert_model_file
– used only in convert_model task
– output filename of converted model
– Note: can be used only in CLI version

### 6.5 Objective Parameters

- objective_seed, default = 5, type = int
– used only in rank_xendcg objective
– random seed for objectives, if random process is needed

- num_class, default = 1, type = int, aliases: num_classes, constraints: num_class > 0
– used only in multi-class classification application

- is_unbalance, default = false, type = bool, aliases: unbalance, unbalanced_sets
– used only in binary and multiclassova applications
– set this to true if training data are unbalanced
– **Note:** while enabling this should increase the overall performance metric of your model, it will also result in poor estimates of the individual class probabilities

– **Note:** this parameter cannot be used at the same time with `scale_pos_weight`, choose only one of them

- `scale_pos_weight`, default = 1.0, type = double, constraints: `scale_pos_weight > 0.0`
  - used only in binary and multiclassova applications
  - weight of labels with positive class
  - **Note:** while enabling this should increase the overall performance metric of your model, it will also result in poor estimates of the individual class probabilities
  - **Note:** this parameter cannot be used at the same time with `is_unbalance`, choose only one of them

- `sigmoid`, default = 1.0, type = double, constraints: `sigmoid > 0.0`
  - used only in binary and multiclassova classification and in lambdarank applications
  - parameter for the sigmoid function

- `boost_from_average`, default = true, type = bool
  - used only in regression, binary, multiclassova and cross-entropy applications
  - adjusts initial score to the mean of labels for faster convergence

- `reg_sqrt`, default = false, type = bool
  - used only in regression application
  - used to fit `sqrt(label)` instead of original values and prediction result will be also automatically converted to `prediction^2`
  - might be useful in case of large-range labels

- `alpha`, default = 0.9, type = double, constraints: `alpha > 0.0`
  - used only in huber and quantile regression applications
  - parameter for Huber loss and Quantile regression

- `fair_c`, default = 1.0, type = double, constraints: `fair_c > 0.0`
  - used only in fair regression application
  - parameter for Fair loss

- `poisson_max_delta_step`, default = 0.7, type = double, constraints: `poisson_max_delta_step > 0.0`
  - used only in poisson regression application
  - parameter for Poisson regression to safeguard optimization

- `tweedie_variance_power`, default = 1.5, type = double, constraints: `1.0 <= tweedie_variance_power < 2.0`
  - used only in tweedie regression application
  - set this closer to 2 to shift towards a Gamma distribution
  - set this closer to 1 to shift towards a Poisson distribution
• lambdarank_truncation_level, default = 30, type = int, constraints: lambdarank_truncation_level > 0
  – used only in lambdarank application
  – controls the number of top-results to focus on during training, refer to “truncation level” in the Sec. 3 of LambdaMART paper
  – this parameter is closely related to the desirable cutoff k in the metric NDCG@k that we aim at optimizing the ranker for. The optimal setting for this parameter is likely to be slightly higher than k (e.g., k + 3) to include more pairs of documents to train on, but perhaps not too high to avoid deviating too much from the desired target metric NDCG@k

• lambdarank_norm, default = true, type = bool
  – used only in lambdarank application
  – set this to true to normalize the lambdas for different queries, and improve the performance for unbalanced data
  – set this to false to enforce the original lambdarank algorithm

• label_gain, default = 0,1,3,7,15,31,63,...,2^30-1, type = multi-double
  – used only in lambdarank application
  – relevant gain for labels. For example, the gain of label 2 is 3 in case of default label gains
  – separate by ,

### 6.6 Metric Parameters

• metric, default = "", type = multi-enum, aliases: metrics,metric_types
  – metric(s) to be evaluated on the evaluation set(s)
    * "" (empty string or not specified) means that metric corresponding to specified objective will be used (this is possible only for pre-defined objective functions, otherwise no evaluation metric will be added)
    * "None" (string, not a None value) means that no metric will be registered, aliases: na, null, custom
    * l1, absolute loss, aliases: mean_absolute_error, mae, regression_l1
    * l2, square loss, aliases: mean_squared_error, mse, regression_l2, regression
    * rmse, root square loss, aliases: root_mean_squared_error, l2_root
    * quantile, Quantile regression
    * mape, MAPE loss, aliases: mean_absolute_percentage_error
    * huber, Huber loss
    * fair, Fair loss
    * poisson, negative log-likelihood for Poisson regression
    * gamma, negative log-likelihood for Gamma regression
    * gamma_deviance, residual deviance for Gamma regression
    * tweedie, negative log-likelihood for Tweedie regression
* ndcg, NDCG, aliases: lambdarank, rank_xendcg, xendcg, xe_ndcg, xe_ndcg_mart, xendcg_mart
* map, MAP, aliases: mean_average_precision
* auc, AUC
* average_precision, average precision score
* binary_logloss, log loss, aliases: binary
* binary_error, for one sample: 0 for correct classification, 1 for error classification
* auc_mu, AUC-mu
* multi_logloss, log loss for multi-class classification, aliases: multiclass, softmax, multiclass_ova, ova, ovr
* multi_error, error rate for multi-class classification
* cross_entropy, cross-entropy (with optional linear weights), aliases: xentropy
* cross_entropy_lambda, “intensity-weighted” cross-entropy, aliases: xentlambda
* kullback_leibler, Kullback-Leibler divergence, aliases: kldiv

  - support multiple metrics, separated by ,

* metric_freq, default = 1, type = int, aliases: output_freq, constraints: metric_freq > 0
  - frequency for metric output
  - Note: can be used only in CLI version

* is_provide_training_metric, default = false, type = bool, aliases: training_metric, is_training_metric, train_metric
  - set this to true to output metric result over training dataset
  - Note: can be used only in CLI version

* eval_at, default = 1,2,3,4,5, type = multi-int, aliases: ndcg_eval_at, ndcg_at, map_eval_at, map_at
  - used only with ndcg and map metrics
  - NDCG and MAP evaluation positions, separated by ,

* multi_error_top_k, default = 1, type = int, constraints: multi_error_top_k > 0
  - used only with multi_error metric
  - threshold for top-k multi-error metric
  - the error on each sample is 0 if the true class is among the top multi_error_top_k predictions, and 1 otherwise
    * more precisely, the error on a sample is 0 if there are at least num_classes - multi_error_top_k predictions strictly less than the prediction on the true class
    - when multi_error_top_k=1 this is equivalent to the usual multi-error metric

* auc_mu_weights, default = None, type = multi-double
  - used only with auc_mu metric
  - list representing flattened matrix (in row-major order) giving loss weights for classification errors
  - list should have n * n elements, where n is the number of classes
the matrix co-ordinate \([i, j]\) should correspond to the \(i \times n + j\)-th element of the list
- if not specified, will use equal weights for all classes

### 6.7 Network Parameters

- **num_machines**, default = 1, type = int, aliases: num_machine, constraints: num_machines > 0
  - the number of machines for distributed learning application
  - this parameter is needed to be set in both socket and mpi versions
- **local_listen_port**, default = 12400 (random for Dask-package), type = int, aliases: local_port, port, constraints: local_listen_port > 0
  - TCP listen port for local machines
  - **Note**: don’t forget to allow this port in firewall settings before training
- **time_out**, default = 120, type = int, constraints: time_out > 0
  - socket time-out in minutes
- **machine_list_filename**, default = "", type = string, aliases: machine_list_file, machine_list, mlist
  - path of file that lists machines for this distributed learning application
  - each line contains one IP and one port for one machine. The format is ip port (space as a separator)
  - **Note**: can be used only in CLI version
- **machines**, default = "", type = string, aliases: workers, nodes
  - list of machines in the following format: ip1:port1,ip2:port2

### 6.8 GPU Parameters

- **gpu_platform_id**, default = -1, type = int
  - OpenCL platform ID. Usually each GPU vendor exposes one OpenCL platform
  - -1 means the system-wide default platform
  - **Note**: refer to GPU Targets for more details
- **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
  - -1 means the default device in the selected platform
  - **Note**: refer to GPU Targets for more details
- **gpu_use_dp**, default = false, type = bool
  - set this to true to use double precision math on GPU (by default single precision is used)
  - **Note**: can be used only in OpenCL implementation, in CUDA implementation only double precision is currently supported
- **num_gpu**, default = 1, type = int, constraints: num_gpu > 0
  - number of GPUs
6.9 Others

6.9.1 Continued Training with Input Score

LightGBM supports continued training with initial scores. It uses an additional file to store these initial scores, like the following:

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

It means the initial score of the first data row 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 placed in the same folder as the data file. In this case, LightGBM will auto load initial score file if it exists.

6.9.2 Weight Data

LightGBM supports 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 the first data row 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 placed in the same folder as the data file. In this case, LightGBM will load the weight file automatically if it exists.

Also, you can include weight column in your data file. Please refer to the `weight_column` parameter in above.

6.9.3 Query Data

For learning to rank, it needs query information for training data.

LightGBM uses an additional file to store query data, like the following:

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

For wrapper libraries like in Python and R, this information can also be provided as an array-like via the Dataset parameter group.

```
[27, 18, 67, ...]
```
For example, if you have a 112-document dataset with \texttt{group = [27, 18, 67]}, that means that you have 3 groups, where the first 27 records are in the first group, records 28-45 are in the second group, and records 46-112 are in the third group.

\textbf{Note:} data should be ordered by the query.

If the name of data file is \texttt{train.txt}, the query file should be named as \texttt{train.txt.query} and placed in the same folder as the data file. In this case, LightGBM will load the query file automatically if it exists.

Also, you can include query/group id column in your data file. Please refer to the \texttt{group_column} parameter in above.
This page contains parameters tuning guides for different scenarios.

**List of other helpful links**

- Parameters
- Python API
- FLAML for automated hyperparameter tuning
- Optuna for automated hyperparameter tuning

### 7.1 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 converge 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 $\text{num\_leaves} = 2^{\text{max\_depth}}$ to obtain the same number of leaves as depth-wise tree. However, this simple conversion is not good in practice. The reason is that a leaf-wise tree is typically much deeper than a depth-wise tree for a fixed number of leaves. Unconstrained depth can induce over-fitting. Thus, when trying to tune the \text{num\_leaves}, we should let it be smaller than $2^{\text{max\_depth}}$. For example, when the \text{max\_depth}=7 the depth-wise tree can get good accuracy, but setting \text{num\_leaves} to 127 may cause over-fitting, and setting it to 70 or 80 may get better accuracy than depth-wise.

2. **min\_data\_in\_leaf.** This is a very important parameter to prevent over-fitting in a leaf-wise tree. Its optimal value depends on the number of training samples and \text{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 \text{max\_depth} to limit the tree depth explicitly.
7.2 For Faster Speed

7.2.1 Add More Computational Resources

On systems where it is available, LightGBM uses OpenMP to parallelize many operations. The maximum number of threads used by LightGBM is controlled by the parameter `num_threads`. By default, this will defer to the default behavior of OpenMP (one thread per real CPU core or the value in environment variable `OMP_NUM_THREADS`, if it is set). For best performance, set this to the number of real CPU cores available.

You might be able to achieve faster training by moving to a machine with more available CPU cores.

Using distributed (multi-machine) training might also reduce training time. See the Distributed Learning Guide for details.

7.2.2 Use a GPU-enabled version of LightGBM

You might find that training is faster using a GPU-enabled build of LightGBM. See the GPU Tutorial for details.

7.2.3 Grow Shallower Trees

The total training time for LightGBM increases with the total number of tree nodes added. LightGBM comes with several parameters that can be used to control the number of nodes per tree.

The suggestions below will speed up training, but might hurt training accuracy.

**Decrease `max_depth`**

This parameter is an integer that controls the maximum distance between the root node of each tree and a leaf node. Decrease `max_depth` to reduce training time.

**Decrease `num_leaves`**

LightGBM adds nodes to trees based on the gain from adding that node, regardless of depth. This figure from the feature documentation illustrates the process.

![Leaf-wise tree growth](image)

Because of this growth strategy, it isn’t straightforward to use `max_depth` alone to limit the complexity of trees. The `num_leaves` parameter sets the maximum number of nodes per tree. Decrease `num_leaves` to reduce training time.
Increase \texttt{min\_gain\_to\_split}

When adding a new tree node, LightGBM chooses the split point that has the largest gain. Gain is basically the reduction in training loss that results from adding a split point. By default, LightGBM sets \texttt{min\_gain\_to\_split} to 0.0, which means “there is no improvement that is too small”. However, in practice you might find that very small improvements in the training loss don’t have a meaningful impact on the generalization error of the model. Increase \texttt{min\_gain\_to\_split} to reduce training time.

Increase \texttt{min\_data\_in\_leaf} and \texttt{min\_sum\_hessian\_in\_leaf}

Depending on the size of the training data and the distribution of features, it’s possible for LightGBM to add tree nodes that only describe a small number of observations. In the most extreme case, consider the addition of a tree node that only a single observation from the training data falls into. This is very unlikely to generalize well, and probably is a sign of overfitting.

This can be prevented indirectly with parameters like \texttt{max\_depth} and \texttt{num\_leaves}, but LightGBM also offers parameters to help you directly avoid adding these overly-specific tree nodes.

- \texttt{min\_data\_in\_leaf}: Minimum number of observations that must fall into a tree node for it to be added.
- \texttt{min\_sum\_hessian\_in\_leaf}: Minimum sum of the Hessian (second derivative of the objective function evaluated for each observation) for observations in a leaf. For some regression objectives, this is just the minimum number of records that have to fall into each node. For classification objectives, it represents a sum over a distribution of probabilities. See this Stack Overflow answer for a good description of how to reason about values of this parameter.

7.2.4 Grow Less Trees

Decrease \texttt{num\_iterations}

The \texttt{num\_iterations} parameter controls the number of boosting rounds that will be performed. Since LightGBM uses decision trees as the learners, this can also be thought of as “number of trees”.

If you try changing \texttt{num\_iterations}, change the \texttt{learning\_rate} as well. \texttt{learning\_rate} will not have any impact on training time, but it will impact the training accuracy. As a general rule, if you reduce \texttt{num\_iterations}, you should increase \texttt{learning\_rate}.

Choosing the right value of \texttt{num\_iterations} and \texttt{learning\_rate} is highly dependent on the data and objective, so these parameters are often chosen from a set of possible values through hyperparameter tuning.

Decrease \texttt{num\_iterations} to reduce training time.

Use Early Stopping

If early stopping is enabled, after each boosting round the model’s training accuracy is evaluated against a validation set that contains data not available to the training process. That accuracy is then compared to the accuracy as of the previous boosting round. If the model’s accuracy fails to improve for some number of consecutive rounds, LightGBM stops the training process.

That “number of consecutive rounds” is controlled by the parameter \texttt{early\_stopping\_rounds}. For example, \texttt{early\_stopping\_rounds=1} says “the first time accuracy on the validation set does not improve, stop training”.

Set \texttt{early\_stopping\_rounds} and provide a validation set to possibly reduce training time.
7.2.5 Consider Fewer Splits

The parameters described in previous sections control how many trees are constructed and how many nodes are constructed per tree. Training time can be further reduced by reducing the amount of time needed to add a tree node to the model.

The suggestions below will speed up training, but might hurt training accuracy.

Enable Feature Pre-Filtering When Creating Dataset

By default, when a LightGBM `Dataset` object is constructed, some features will be filtered out based on the value of `min_data_in_leaf`.

For a simple example, consider a 1000-observation dataset with a feature called `feature_1`. `feature_1` takes on only two values: 25.0 (995 observations) and 50.0 (5 observations). If `min_data_in_leaf = 10`, there is no split for this feature which will result in a valid split at least one of the leaf nodes will only have 5 observations.

Instead of reconsidering this feature and then ignoring it every iteration, LightGBM filters this feature out at before training, when the `Dataset` is constructed.

If this default behavior has been overridden by setting `feature_pre_filter=False`, set `feature_pre_filter=True` to reduce training time.

Decrease `max_bin` or `max_bin_by_feature` When Creating Dataset

LightGBM training buckets continuous features into discrete bins to improve training speed and reduce memory requirements for training. This binning is done one time during `Dataset` construction. The number of splits considered when adding a node is \(O(\#\text{feature} \times \#\text{bin})\), so reducing the number of bins per feature can reduce the number of splits that need to be evaluated.

`max_bin` is controls the maximum number of bins that features will bucketed into. It is also possible to set this maximum feature-by-feature, by passing `max_bin_by_feature`.

Reduce `max_bin` or `max_bin_by_feature` to reduce training time.

Increase `min_data_in_bin` When Creating Dataset

Some bins might contain a small number of observations, which might mean that the effort of evaluating that bin’s boundaries as possible split points isn’t likely to change the final model very much. You can control the granularity of the bins by setting `min_data_in_bin`.

Increase `min_data_in_bin` to reduce training time.

Decrease `feature_fraction`

By default, LightGBM considers all features in a `Dataset` during the training process. This behavior can be changed by setting `feature_fraction` to a value > 0 and <= 1.0. Setting `feature_fraction` to 0.5, for example, tells LightGBM to randomly select 50% of features at the beginning of constructing each tree. This reduces the total number of splits that have to be evaluated to add each tree node.

Decrease `feature_fraction` to reduce training time.
**Decrease max_cat_threshold**

LightGBM uses a custom approach for finding optimal splits for categorical features. In this process, LightGBM explores splits that break a categorical feature into two groups. These are sometimes called “k-vs.-rest” splits. Higher max_cat_threshold values correspond to more split points and larger possible group sizes to search.

Decrease max_cat_threshold to reduce training time.

**7.2.6 Use Less Data**

**Use Bagging**

By default, LightGBM uses all observations in the training data for each iteration. It is possible to instead tell LightGBM to randomly sample the training data. This process of training over multiple random samples without replacement is called “bagging”.

Set bagging_freq to an integer greater than 0 to control how often a new sample is drawn. Set bagging_fraction to a value > 0.0 and < 1.0 to control the size of the sample. For example,{"bagging_freq": 5, "bagging_fraction": 0.75} tells LightGBM “re-sample without replacement every 5 iterations, and draw samples of 75% of the training data”.

Decrease bagging_fraction to reduce training time.

**7.2.7 Save Constructed Datasets with save_binary**

This only applies to the LightGBM CLI. If you pass parameter save_binary, the training dataset and all validations sets will be saved in a binary format understood by LightGBM. This can speed up training next time, because binning and other work done when constructing a Dataset does not have to be re-done.

**7.3 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

**7.4 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` for regularization
• Try `max_depth` to avoid growing deep tree
• Try `extra_trees`
• Try increasing `path_smooth`
Copyright Copyright (c) 2016 Microsoft Corporation. All rights reserved. Licensed under the MIT License. See LICENSE file in the project root for license information.

Note To avoid type conversion on large data, the most of our exposed interface supports both float32 and float64, except the following:

1. gradient and Hessian;
2. current score for training and validation data.

The reason is that they are called frequently, and the type conversion on them may be time-cost.

Defines

C_API_DTYPE_FLOAT32 (0)
float32 (single precision float).

C_API_DTYPE_FLOAT64 (1)
float64 (double precision float).

C_API_DTYPE_INT32 (2)
int32.

C_API_DTYPE_INT64 (3)
int64.

C_API_FEATURE_IMPORTANCE_GAIN (1)
Gain type of feature importance.

C_API_FEATURE_IMPORTANCE_SPLIT (0)
Split type of feature importance.

C_API_MATRIX_TYPE_CSC (1)
CSC sparse matrix type.

C_API_MATRIX_TYPE_CSR (0)
CSR sparse matrix type.

C_API_PREDICT_CONTRIB (3)
Predict feature contributions (SHAP values).

C_API_PREDICT_LEAF_INDEX (2)
Predict leaf index.

C_API_PREDICT_NORMAL (0)
Normal prediction, with transform (if needed).
C API PREDICT_RAW_SCORE (1)
  Predict raw score.

THREAD_LOCAL thread_local
  Thread local specifier.

TypeDefs

typedef void *BoosterHandle
  Handle of booster.

typedef void *DatasetHandle
  Handle of dataset.

typedef void *FastConfigHandle
  Handle of FastConfig.

Functions

static char *LastErrorMsg ()
  Handle of error message.

Return Error message

LIGHTGBM_C_EXPORT int LGBM_BoosterAddValidData (BoosterHandle handle, const DatasetHandle valid_data)
  Add new validation data to booster.

Return 0 when succeed, -1 when failure happens

Parameters
  • handle: Handle of booster
  • valid_data: Validation dataset

LIGHTGBM_C_EXPORT int LGBM_BoosterCalcNumPredict (BoosterHandle handle, int num_row, int predict_type, int start_iteration, int num_iteration, int64_t *out_len)
  Get number of predictions.

Return 0 when succeed, -1 when failure happens

Parameters
  • handle: Handle of booster
  • num_row: Number of rows
  • predict_type: What should be predicted
    - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
    - C_API_PREDICT_RAW_SCORE: raw score;
    - C_API_PREDICT_LEAF_INDEX: leaf index;
    - C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
• start_iteration: Start index of the iteration to predict
• num_iteration: Number of iterations for prediction, <= 0 means no limit
• [out] out_len: Length of prediction

LIGHTGBM_C_EXPORT int LGBM_BoosterCreate(const DatasetHandle train_data, const char *parameters, BoosterHandle *out)

Create a new boosting learner.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- train_data: Training dataset
- parameters: Parameters in format ‘key1=value1 key2=value2’
- [out] out: Handle of created booster

LIGHTGBM_C_EXPORT int LGBM_BoosterCreateFromModelfile(const char *filename, int *out_num_iterations, BoosterHandle *out)

Load an existing booster from model file.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- filename: Filename of model
- [out] out_num_iterations: Number of iterations of this booster
- [out] out: Handle of created booster

LIGHTGBM_C_EXPORT int LGBM_BoosterDumpModel(BoosterHandle handle, int start_iteration, int num_iteration, int feature_importance_type, int64_t buffer_len, int64_t *out_len, char *out_str)

Dump model to JSON.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- handle: Handle of booster
- start_iteration: Start index of the iteration that should be dumped
- num_iteration: Index of the iteration that should be dumped, <= 0 means dump all
- feature_importance_type: Type of feature importance, can be C_API_FEATURE_IMPORTANCE_SPLIT or C_API_FEATURE_IMPORTANCE_GAIN
- buffer_len: String buffer length, if buffer_len < out_len, you should re-allocate buffer
- [out] out_len: Actual output length
- [out] out_str: JSON format string of model, should pre-allocate memory
Get model feature importance.

Return 0 when succeed, -1 when failure happens

Parameters
  • handle: Handle of booster
  • num_iteration: Number of iterations for which feature importance is calculated, <= 0 means use all
  • importance_type: Method of importance calculation:
    – C_API_FEATURE_IMPORTANCE_SPLIT: result contains numbers of times the feature is used in a model;
    – C_API_FEATURE_IMPORTANCE_GAIN: result contains total gains of splits which use the feature
  • [out] out_results: Result array with feature importance

Free space for booster.

Return 0 when succeed, -1 when failure happens

Parameters
  • handle: Handle of booster to be freed

Method corresponding to LGBM_BoosterPredictSparseOutput to free the allocated data.

Return 0 when succeed, -1 when failure happens

Parameters
  • indptr: Pointer to output row headers or column headers to be deallocated
  • indices: Pointer to sparse indices to be deallocated
  • data: Pointer to sparse data space to be deallocated
  • indptr_type: Type of indptr, can be C_API_DTYPE_INT32 or C_API_DTYPE_INT64
  • data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64

Get index of the current boosting iteration.

Return 0 when succeed, -1 when failure happens

Parameters
• **handle**: Handle of booster
  • **[out] out_iteration**: Index of the current boosting iteration

**LIGHTGBM_C_EXPORT int LGBM_BoosterGetEval**(BoosterHandle handle, int data_idx, int *out_len, double *out_results)

Get evaluation for training data and validation data.

**Note**

a. You should call LGBM_BoosterGetEvalNames first to get the names of evaluation datasets.

b. You should pre-allocate memory for out_results, you can get its length by LGBM_BoosterGetEvalCounts.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

• **handle**: Handle of booster

• **data_idx**: Index of data, 0: training data, 1: 1st validation data, 2: 2nd validation data and so on

• **[out] out_len**: Length of output result

• **[out] out_results**: Array with evaluation results

**LIGHTGBM_C_EXPORT int LGBM_BoosterGetEvalCounts**(BoosterHandle handle, int *out_len)

Get number of evaluation datasets.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

• **handle**: Handle of booster

• **out_len**: Total number of evaluation datasets

**LIGHTGBM_C_EXPORT int LGBM_BoosterGetEvalNames**(BoosterHandle handle, const int len, int *out_len, const size_t buffer_len, size_t *out_buffer_len, char **out_strs)

Get names of evaluation datasets.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

• **handle**: Handle of booster

• **len**: Number of char* pointers stored at out_strs. If smaller than the max size, only this many strings are copied

• **[out] out_len**: Total number of evaluation datasets

• **buffer_len**: Size of pre-allocated strings. Content is copied up to buffer_len - 1 and null-terminated

• **[out] out_buffer_len**: String sizes required to do the full string copies

• **[out] out_strs**: Names of evaluation datasets, should pre-allocate memory
LIGHTGBM_C_EXPORT int LGBM_BoosterGetFeatureNames(BoosterHandle handle, const int len, int *out_len, const size_t buffer_len, size_t *out_buffer_len, char **out_strs)

Get names of features.

Return 0 when succeed, -1 when failure happens

Parameters

• handle: Handle of booster
• len: Number of char* pointers stored at out_strs. If smaller than the max size, only this many strings are copied
• [out] out_len: Total number of features
• buffer_len: Size of pre-allocated strings. Content is copied up to buffer_len - 1 and null-terminated
• [out] out_buffer_len: String sizes required to do the full string copies
• [out] out_strs: Names of features, should pre-allocate memory

LIGHTGBM_C_EXPORT int LGBM_BoosterGetLeafValue(BoosterHandle handle, int tree_idx, int leaf_idx, double *out_val)

Get leaf value.

Return 0 when succeed, -1 when failure happens

Parameters

• handle: Handle of booster
• tree_idx: Index of tree
• leaf_idx: Index of leaf
• [out] out_val: Output result from the specified leaf

LIGHTGBM_C_EXPORT int LGBM_BoosterGetLinear(BoosterHandle handle, bool *out)

Get boolean representing whether booster is fitting linear trees.

Return 0 when succeed, -1 when failure happens

Parameters

• handle: Handle of booster
• [out] out: The address to hold linear trees indicator

LIGHTGBM_C_EXPORT int LGBM_BoosterGetLowerBoundValue(BoosterHandle handle, double *out_results)

Get model lower bound value.

Return 0 when succeed, -1 when failure happens

Parameters

• handle: Handle of booster
• [out] out_results: Result pointing to min value

LIGHTGBM_C_EXPORT int LGBM_BoosterGetNumClasses(BoosterHandle handle, int *out_len)
Get number of classes.

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of booster
• [out] out_len: Number of classes

LIGHTGBM_C_EXPORT int LGBM_BoosterGetNumFeature(BoosterHandle handle, int *out_len)
Get number of features.

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of booster
• [out] out_len: Total number of features

LIGHTGBM_C_EXPORT int LGBM_BoosterGetNumPredict(BoosterHandle handle, int data_idx, int64_t *out_len)
Get number of predictions for training data and validation data (this can be used to support customized evaluation functions).

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of booster
• data_idx: Index of data, 0: training data, 1: 1st validation data, 2: 2nd validation data and so on
• [out] out_len: Number of predictions

LIGHTGBM_C_EXPORT int LGBM_BoosterGetPredict(BoosterHandle handle, int data_idx, int64_t *out_len, double *out_result)
Get prediction for training data and validation data.

Note You should pre-allocate memory for out_result, its length is equal to num_class * num_data.

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of booster
• data_idx: Index of data, 0: training data, 1: 1st validation data, 2: 2nd validation data and so on
• [out] out_len: Length of output result
• [out] out_result: Pointer to array with predictions

LIGHTGBM_C_EXPORT int LGBM_BoosterGetUpperBoundValue(BoosterHandle handle, double *out_results)
Get model upper bound value.
**Return** 0 when succeed, -1 when failure happens

**Parameters**
- handle: Handle of booster
- [out] out_results: Result pointing to max value

LIGHTGBM_C_EXPORT int LGBM_BoosterLoadModelFromString(const char *model_str, int *out_num_iterations, BoosterHandle *out)

Load an existing booster from string.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- model_str: Model string
- [out] out_num_iterations: Number of iterations of this booster
- [out] out: Handle of created booster

LIGHTGBM_C_EXPORT int LGBM_BoosterMerge(BoosterHandle handle, BoosterHandle other_handle)

Merge model from other_handle into handle.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- handle: Handle of booster, will merge another booster into this one
- other_handle: Other handle of booster

LIGHTGBM_C_EXPORT int LGBM_BoosterNumberOfTotalModel(BoosterHandle handle, int *out_models)

Get number of weak sub-models.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- handle: Handle of booster
- [out] out_models: Number of weak sub-models

LIGHTGBM_C_EXPORT int LGBM_BoosterNumModelPerIteration(BoosterHandle handle, int *out_tree_per_iteration)

Get number of trees per iteration.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- handle: Handle of booster
- [out] out_tree_per_iteration: Number of trees per iteration
LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForCSC(BoosterHandle handle, const void *col_ptr, int col_ptr_type, const void *data, const int32_t *indices, int data_type, int ncol_ptr, int64_t nelem, int64_t num_row, int predict_type, int start_iteration, int num_iteration, const char *parameter, int64_t *out_len, double *out_result)

Make prediction for a new dataset in CSC format.

**Note** You should pre-allocate memory for `out_result`:
- for normal and raw score, its length is equal to `num_class * num_data`;
- for leaf index, its length is equal to `num_class * num_data * num_iteration`;
- for feature contributions, its length is equal to `num_class * num_data * (num_feature + 1)`.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- `handle`: Handle of booster
- `col_ptr`: Pointer to column headers
- `col_ptr_type`: Type of `col_ptr`, can be C_API_DTYPE_INT32 or C_API_DTYPE_INT64
- `indices`: Pointer to row indices
- `data`: Pointer to the data space
- `data_type`: Type of `data` pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- `ncol_ptr`: Number of columns in the matrix + 1
- `nelem`: Number of nonzero elements in the matrix
- `num_row`: Number of rows
- `predict_type`: What should be predicted
  - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
  - C_API_PREDICT_RAW_SCORE: raw score;
  - C_API_PREDICT_LEAF_INDEX: leaf index;
  - C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
- `start_iteration`: Start index of the iteration to predict
- `num_iteration`: Number of iteration for prediction, <= 0 means no limit
- `parameter`: Other parameters for prediction, e.g. early stopping for prediction
- `[out] out_len`: Length of output result
- `[out] out_result`: Pointer to array with predictions
**LightGBM, Release 3.2.0.99**

LightGBM, Release 3.2.0.99

LIGHTGBM_C_EXPORT int **LGBM_BoosterPredictForCSR**(BoosterHandle **handle**, const void **indptr**, int **indptr_type**, const int32_t **indices**, const void **data**, int **data_type**, int64_t **nindptr**, int64_t **nelem**, int64_t **num_col**, int **predict_type**, int **start_iteration**, int **num_iteration**, const char **parameter**, int64_t **out_len**, double **out_result**)

Make prediction for a new dataset in CSR format.

**Note** You should pre-allocate memory for **out_result**:
- for normal and raw score, its length is equal to num_class \* num_data;
- for leaf index, its length is equal to num_class \* num_data \* num_iteration;
- for feature contributions, its length is equal to num_class \* num_data \* (num_feature + 1).

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- **handle**: Handle of booster
- **indptr**: Pointer to row headers
- **indptr_type**: Type of indptr, can be C_API_DTYPE_INT32 or C_API_DTYPE_INT64
- **indices**: Pointer to column indices
- **data**: Pointer to the data space
- **data_type**: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- **nindptr**: Number of rows in the matrix + 1
- **nelem**: Number of nonzero elements in the matrix
- **num_col**: Number of columns
- **predict_type**: What should be predicted
  - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
  - C_API_PREDICT_RAW_SCORE: raw score;
  - C_API_PREDICT_LEAF_INDEX: leaf index;
  - C_API_PREDICT_CONTRIBUTION: feature contributions (SHAP values)
- **start_iteration**: Start index of the iteration to predict
- **num_iteration**: Number of iterations for prediction, <= 0 means no limit
- **parameter**: Other parameters for prediction, e.g. early stopping for prediction
- [out] **out_len**: Length of output result
- [out] **out_result**: Pointer to array with predictions
Make prediction for a new dataset in CSR format. This method re-uses the internal predictor structure from previous calls and is optimized for single row invocation.

**Note** You should pre-allocate memory for `out_result`:
- for normal and raw score, its length is equal to `num_class * num_data`;
- for leaf index, its length is equal to `num_class * num_data * num_iteration`;
- for feature contributions, its length is equal to `num_class * num_data * (num_feature + 1)`.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- `handle`: Handle of booster
- `indptr`: Pointer to row headers
- `indptr_type`: Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- `indices`: Pointer to column indices
- `data`: Pointer to the data space
- `data_type`: Type of data pointer, can be `C_API_DTYPEFLOAT32` or `C_API_DTYPE_FLOAT64`
- `nindptr`: Number of rows in the matrix + 1
- `nelem`: Number of nonzero elements in the matrix
- `num_col`: Number of columns
- `predict_type`: What should be predicted
  - `C_API_PREDICT_NORMAL`: normal prediction, with transform (if needed);
  - `C_API_PREDICT_RAW_SCORE`: raw score;
  - `C_API_PREDICT_LEAF_INDEX`: leaf index;
  - `C_API_PREDICT_CONTRIB`: feature contributions (SHAP values)
- `start_iteration`: Start index of the iteration to predict
- `num_iteration`: Number of iterations for prediction, <= 0 means no limit
- `parameter`: Other parameters for prediction, e.g. early stopping for prediction
- `[out] out_len`: Length of output result
• [out] out_result: Pointer to array with predictions

```
LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForCSRSingleRowFast(FastConfigHandle fastConfig_handle,
    const void *indptr, const int indptr_type, const int32_t *indices,
    const void *data,
    const int64_t nindptr, const int64_t nelem,
    int64_t *out_len,
    double *out_result)
```

Faster variant of LGBM_BoosterPredictForCSRSingleRow.

Score single rows after setup with LGBM_BoosterPredictForCSRSingleRowFastInit.

By removing the setup steps from this call extra optimizations can be made like initializing the config only once, instead of once per call.

**Note** Setting up the number of threads is only done once at LGBM_BoosterPredictForCSRSingleRowFastInit instead of at each prediction. If you use a different number of threads in other calls, you need to start the setup process over, or that number of threads will be used for these calls as well.

**Note** You should pre-allocate memory for `out_result`:
- for normal and raw score, its length is equal to `num_class * num_data`;
- for leaf index, its length is equal to `num_class * num_data * num_iteration`;
- for feature contributions, its length is equal to `num_class * num_data * (num_feature + 1)`.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- `fastConfig_handle`: FastConfig object handle returned by LGBM_BoosterPredictForCSRSingleRowFastInit
- `indptr`: Pointer to row headers
- `indptr_type`: Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- `indices`: Pointer to column indices
- `data`: Pointer to the data space
- `nindptr`: Number of rows in the matrix + 1
- `nelem`: Number of nonzero elements in the matrix
- [out] `out_len`: Length of output result
- [out] `out_result`: Pointer to array with predictions
**LGBM_BoosterPredictForCSRSingleRowFastInit**

```c
LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForCSRSingleRowFastInit(BoosterHandle handle,
                                                                  const int predict_type,
                                                                  const int start_iteration,
                                                                  const int num_iteration,
                                                                  const int data_type,
                                                                  const int64_t num_col,
                                                                  const char *parameter,
                                                                  FastConfigHandle *out_fastConfig)
```

Initialize and return a FastConfigHandle for use with LGBM_BoosterPredictForCSRSingleRowFast.

Release the FastConfig by passing its handle to LGBM_FastConfigFree when no longer needed.

**Return** 0 when it succeeds, -1 when failure happens

**Parameters**

- **handle**: Booster handle
- **predict_type**: What should be predicted
  - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
  - C_API_PREDICT_RAW_SCORE: raw score;
  - C_API_PREDICT_LEAF_INDEX: leaf index;
  - C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
- **start_iteration**: Start index of the iteration to predict
- **num_iteration**: Number of iterations for prediction, <= 0 means no limit
- **data_type**: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- **num_col**: Number of columns
- **parameter**: Other parameters for prediction, e.g. early stopping for prediction
- **[out] out_fastConfig**: FastConfig object with which you can call LGBM_BoosterPredictForCSRSingleRowFast

**LGBM_BoosterPredictForFile**

```c
LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForFile(BoosterHandle handle,
                                                  const char *filename, int data_has_header,
                                                  int predict_type, int start_iteration, int
                                                  num_iteration, const char *parameter, const
                                                  char *result_filename)
```

Make prediction for file.

**Return** 0 when succeed, -1 when failure happens
Parameters

- handle: Handle of booster
- data_filename: Filename of file with data
- data_has_header: Whether file has header or not
- predict_type: What should be predicted
  - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
  - C_API_PREDICT_RAW_SCORE: raw score;
  - C_API_PREDICT_LEAF_INDEX: leaf index;
  - C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
- start_iteration: Start index of the iteration to predict
- num_iteration: Number of iterations for prediction. <= 0 means no limit
- parameter: Other parameters for prediction, e.g. early stopping for prediction
- result_filename: Filename of result file in which predictions will be written

```
LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForMat (BoosterHandle handle, const void *data, int data_type, int32_t nrow, int32_t ncol, int is_row_major, int predict_type, int start_iteration, int num_iteration, const char *parameter, int64_t*out_len, double*out_result)
```

Make prediction for a new dataset.

**Note** You should pre-allocate memory for `out_result`:

- for normal and raw score, its length is equal to `num_class * num_data`;
- for leaf index, its length is equal to `num_class * num_data * num_iteration`;
- for feature contributions, its length is equal to `num_class * num_data * (num_feature + 1)`.

**Return** 0 when succeed, -1 when failure happens

Parameters

- handle: Handle of booster
- data: Pointer to the data space
- data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- nrow: Number of rows
- ncol: Number of columns
- is_row_major: 1 for row-major, 0 for column-major
- predict_type: What should be predicted
  - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
  - C_API_PREDICT_RAW_SCORE: raw score;
  - C_API_PREDICT_LEAF_INDEX: leaf index;
LightGBM, Release 3.2.0.99

- C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
- start_iteration: Start index of the iteration to predict
- num_iteration: Number of iteration for prediction, <= 0 means no limit
- parameter: Other parameters for prediction, e.g. early stopping for prediction
- [out] out_len: Length of output result
- [out] out_result: Pointer to array with predictions

LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForMats(BoosterHandle handle, const void **data, int data_type, int32_t nrow, int32_t ncol, int predict_type, int start_iteration, int num_iteration, const char *parameter, int64_t *out_len, double *out_result)

Make prediction for a new dataset presented in a form of array of pointers to rows.

Note You should pre-allocate memory for out_result:
- for normal and raw score, its length is equal to num_class * num_data;
- for leaf index, its length is equal to num_class * num_data * num_iteration;
- for feature contributions, its length is equal to num_class * num_data * (num_feature + 1).

Return 0 when succeed, -1 when failure happens

Parameters
- handle: Handle of booster
- data: Pointer to the data space
- data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- nrow: Number of rows
- ncol: Number columns
- predict_type: What should be predicted
  - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
  - C_API_PREDICT_RAW_SCORE: raw score;
  - C_API_PREDICT_LEAF_INDEX: leaf index;
  - C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
- start_iteration: Start index of the iteration to predict
- num_iteration: Number of iteration for prediction, <= 0 means no limit
- parameter: Other parameters for prediction, e.g. early stopping for prediction
- [out] out_len: Length of output result
- [out] out_result: Pointer to array with predictions
LightGBM, Release 3.2.0.99

LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForMatSingleRow (BoosterHandle handle, const void *data, int data_type, int ncol, int is_row_major, int predict_type, int start_iteration, int num_iteration, const char *parameter, int64_t *out_len, double *out_result)

Make prediction for a new dataset. This method re-uses the internal predictor structure from previous calls and is optimized for single row invocation.

Note You should pre-allocate memory for out_result:
- for normal and raw score, its length is equal to num_class * num_data;
- for leaf index, its length is equal to num_class * num_data * num_iteration;
- for feature contributions, its length is equal to num_class * num_data * (num_feature + 1).

Return 0 when succeed, -1 when failure happens

Parameters
- handle: Handle of booster
- data: Pointer to the data space
- data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- ncol: Number columns
- is_row_major: 1 for row-major, 0 for column-major
- predict_type: What should be predicted
  - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
  - C_API_PREDICT_RAW_SCORE: raw score;
  - C_API_PREDICT_LEAF_INDEX: leaf index;
  - C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
- start_iteration: Start index of the iteration to predict
- num_iteration: Number of iteration for prediction, <= 0 means no limit
- parameter: Other parameters for prediction, e.g. early stopping for prediction
- [out] out_len: Length of output result
- [out] out_result: Pointer to array with predictions

LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForMatSingleRowFast (FastConfigHandle fastConfig_handle, const void *data, int64_t *out_len, double *out_result)

Faster variant of LGBM_BoosterPredictForMatSingleRow.
Score a single row after setup with `LGBM_BoosterPredictForMatSingleRowFastInit`.

By removing the setup steps from this call extra optimizations can be made like initializing the config only once, instead of once per call.

**Note** Setting up the number of threads is only done once at `LGBM_BoosterPredictForMatSingleRowFastInit` instead of at each prediction. If you use a different number of threads in other calls, you need to start the setup process over, or that number of threads will be used for these calls as well.

**Return** 0 when it succeeds, -1 when failure happens

**Parameters**
- `fastConfig_handle`: FastConfig object handle returned by `LGBM_BoosterPredictForMatSingleRowFastInit`
- `data`: Single-row array data (no other way than row-major form).
- `[out] out_len`: Length of output result
- `[out] out_result`: Pointer to array with predictions

```c
LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForMatSingleRowFastInit
(BoosterHandle handle,
 const int predict_type,
 const int start_iteration,
 const int num_iteration,
 const int data_type,
 const int32_t ncol,
 const char *parameter,
 FastConfigHandle *out_fastConfig)
```

Initialize and return a FastConfigHandle for use with `LGBM_BoosterPredictForMatSingleRowFast`.

Release the FastConfig by passing its handle to `LGBM_FastConfigFree` when no longer needed.

**Return** 0 when it succeeds, -1 when failure happens

**Parameters**
- `handle`: Booster handle
- `predict_type`: What should be predicted
  - `C_API_PREDICT_NORMAL`: normal prediction, with transform (if needed);
  - `C_API_PREDICT_RAW_SCORE`: raw score;
  - `C_API_PREDICT_LEAF_INDEX`: leaf index;
  - `C_API_PREDICT_CONTRIB`: feature contributions (SHAP values)
- `start_iteration`: Start index of the iteration to predict
- `num_iteration`: Number of iterations for prediction, <= 0 means no limit
**data_type**: Type of data pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`

**ncol**: Number of columns

**parameter**: Other parameters for prediction, e.g. early stopping for prediction

**[out]** out_fastConfig: FastConfig object with which you can call `LGBM_BoosterPredictForMatSingleRowFast`

Make sparse prediction for a new dataset in CSR or CSC format. Currently only used for feature contributions.

**Note** The outputs are pre-allocated, as they can vary for each invocation, but the shape should be the same:

- for feature contributions, the shape of sparse matrix will be `num_class * num_data * (num_feature + 1)`. The output `indptr_type` for the sparse matrix will be the same as the given input `indptr_type`. Call `LGBM_BoosterFreePredictSparse` to deallocate resources.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

- **handle**: Handle of booster
- **indptr**: Pointer to row headers for CSR or column headers for CSC
- **indptr_type**: Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- **indices**: Pointer to column indices for CSR or row indices for CSC
- **data**: Pointer to the data space
- **data_type**: Type of data pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`
- **nindptr**: Number of rows in the matrix + 1
- **nelem**: Number of nonzero elements in the matrix
- **num_col_or_row**: Number of columns for CSR or number of rows for CSC
- **predict_type**: What should be predicted, only feature contributions supported currently
  - `C_API_PREDICT_CONTRIB`: feature contributions (SHAP values)
- **start_iteration**: Start index of the iteration to predict
- **num_iteration**: Number of iterations for prediction, <= 0 means no limit
- **parameter**: Other parameters for prediction, e.g. early stopping for prediction
• **matrix_type**: Type of matrix input and output, can be C_API_MATRIX_TYPE_CSR or C_API_MATRIX_TYPE_CSC
• 
  - **[out]** `out_len`: Length of output indices and data
  - **[out]** `out_indptr`: Pointer to output row headers for CSR or column headers for CSC
  - **[out]** `out_indices`: Pointer to sparse column indices for CSR or row indices for CSC
  - **[out]** `out_data`: Pointer to sparse data space

**LIGHTGBM_C_EXPORT int LGBM_BoosterRefit (BoosterHandle handle, const int32_t *leaf_preds, int32_t nrow, int32_t ncol)**

Refit the tree model using the new data (online learning).

**Return** 0 when succeed, -1 when failure happens

**Parameters**

• **handle**: Handle of booster
• **leaf_preds**: Pointer to predicted leaf indices
• **nrow**: Number of rows of `leaf_preds`
• **ncol**: Number of columns of `leaf_preds`

**LIGHTGBM_C_EXPORT int LGBM_BoosterResetParameter (BoosterHandle handle, const char *parameters)**

Reset config for booster.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

• **handle**: Handle of booster
• **parameters**: Parameters in format ‘key1=value1 key2=value2’

**LIGHTGBM_C_EXPORT int LGBM_BoosterResetTrainingData (BoosterHandle handle, const DatasetHandle train_data)**

Reset training data for booster.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

• **handle**: Handle of booster
• **train_data**: Training dataset

**LIGHTGBM_C_EXPORT int LGBM_BoosterRollbackOneIter (BoosterHandle handle)**

Rollback one iteration.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

• **handle**: Handle of booster
LIGHTGBM_C_EXPORT int LGBM_BoosterSaveModel (BoosterHandle handle, int start_iteration, int num_iteration, int feature_importance_type, const char *filename)

Save model into file.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

- handle: Handle of booster
- start_iteration: Start index of the iteration that should be saved
- num_iteration: Index of the iteration that should be saved, <= 0 means save all
- feature_importance_type: Type of feature importance, can be C_API_FEATURE_IMPORTANCE_SPLIT or C_API_FEATURE_IMPORTANCE_GAIN
- filename: The name of the file

LIGHTGBM_C_EXPORT int LGBM_BoosterSaveModelToString (BoosterHandle handle, int start_iteration, int num_iteration, int64_t buffer_len, int64_t *out_len, char *out_str)

Save model to string.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

- handle: Handle of booster
- start_iteration: Start index of the iteration that should be saved
- num_iteration: Index of the iteration that should be saved, <= 0 means save all
- feature_importance_type: Type of feature importance, can be C_API_FEATURE_IMPORTANCE_SPLIT or C_API_FEATURE_IMPORTANCE_GAIN
- buffer_len: String buffer length, if buffer_len < out_len, you should re-allocate buffer
- [out] out_len: Actual output length
- [out] out_str: String of model, should pre-allocate memory

LIGHTGBM_C_EXPORT int LGBM_BoosterSetLeafValue (BoosterHandle handle, int tree_idx, int leaf_idx, double val)

Set leaf value.

**Return** 0 when succeed, -1 when failure happens

**Parameters**

- handle: Handle of booster
- tree_idx: Index of tree
- leaf_idx: Index of leaf
- val: Leaf value
**LightGBM, Release 3.2.0.99**

LIGHTGBM_C_EXPORT int **LGBM_BoosterShuffleModels** *(BoosterHandle handle, int start_iter, int end_iter)*

Shuffle models.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- **handle**: Handle of booster
- **start_iter**: The first iteration that will be shuffled
- **end_iter**: The last iteration that will be shuffled

LIGHTGBM_C_EXPORT int **LGBM_BoosterUpdateOneIter** *(BoosterHandle handle, int *is_finished)*

Update the model for one iteration.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- **handle**: Handle of booster
- **[out] is_finished**: 1 means the update was successfully finished (cannot split any more), 0 indicates failure

LIGHTGBM_C_EXPORT int **LGBM_BoosterUpdateOneIterCustom** *(BoosterHandle handle, const float *grad, const float *hess, int *is_finished)*

Update the model by specifying gradient and Hessian directly (this can be used to support customized loss functions).

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- **handle**: Handle of booster
- **grad**: The first order derivative (gradient) statistics
- **hess**: The second order derivative (Hessian) statistics
- **[out] is_finished**: 1 means the update was successfully finished (cannot split any more), 0 indicates failure

LIGHTGBM_C_EXPORT int **LGBM_DatasetAddFeaturesFrom** *(DatasetHandle target, DatasetHandle source)*

Add features from source to target.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- **target**: The handle of the dataset to add features to
- **source**: The handle of the dataset to take features from
LIGHTGBM_C_EXPORT int LGBM_DatasetCreateByReference(const DatasetHandle reference, int64_t num_total_row, DatasetHandle *out)

Allocate the space for dataset and bucket feature bins according to reference dataset.

Return 0 when succeed, -1 when failure happens

Parameters

- reference: Used to align bin mapper with other dataset
- num_total_row: Number of total rows
- [out] out: Created dataset

LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromCSC(const void *col_ptr, int col_ptr_type, const int32_t *indices, const void *data, int data_type, int64_t ncol_ptr, int64_t nelem, int64_t num_row, const char *parameters, const DatasetHandle reference, DatasetHandle *out)

Create a dataset from CSC format.

Return 0 when succeed, -1 when failure happens

Parameters

- col_ptr: Pointer to column headers
- col_ptr_type: Type of col_ptr, can be C_API_DTYPE_INT32 or C_API_DTYPE_INT64
- indices: Pointer to row indices
- data: Pointer to the data space
- data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- ncol_ptr: Number of columns in the matrix + 1
- nelem: Number of nonzero elements in the matrix
- num_row: Number of rows
- parameters: Additional parameters
- reference: Used to align bin mapper with other dataset, nullptr means it’s not used
- [out] out: Created dataset

LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromCSR(const void *indptr, int indptr_type, const int32_t *indices, const void *data, int data_type, int64_t nindptr, int64_t nelem, int64_t num_col, const char *parameters, const DatasetHandle reference, DatasetHandle *out)

Create a dataset from CSR format.

Return 0 when succeed, -1 when failure happens
Parameters

- `indptr`: Pointer to row headers
- `indptr_type`: Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- `indices`: Pointer to column indices
- `data`: Pointer to the data space
- `data_type`: Type of data pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`
- `nindptr`: Number of rows in the matrix + 1
- `nelem`: Number of nonzero elements in the matrix
- `num_col`: Number of columns
- `parameters`: Additional parameters
- `reference`: Used to align bin mapper with other dataset, `nullptr` means isn’t used
- `[out] out`: Created dataset

```c
LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromCSRFunc
(void *get_row_funptr, int num_rows, int64_t num_col,
const char *parameters, const DatasetHandle reference,
DatasetHandle *out)
```

Create a dataset from CSR format through callbacks.

**Return** 0 when succeed, -1 when failure happens

Parameters

- `get_row_funptr`: Pointer to `std::function<void(int idx,
std::vector<std::pair<int, double>>& ret)>` (called for every row and expected to clear and fill `ret`)
- `num_rows`: Number of rows
- `num_col`: Number of columns
- `parameters`: Additional parameters
- `reference`: Used to align bin mapper with other dataset, `nullptr` means isn’t used
- `[out] out`: Created dataset

```c
LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromFile(const char *filename, const char *parameters, const DatasetHandle reference, DatasetHandle *out)
```

Load dataset from file (like LightGBM CLI version does).

**Return** 0 when succeed, -1 when failure happens

Parameters

- `filename`: The name of the file
- `parameters`: Additional parameters
- `reference`: Used to align bin mapper with other dataset, `nullptr` means isn’t used
LightGBM, Release 3.2.0.99

- [out] out: A loaded dataset

LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromMat(const void *data, int data_type, int32_t nrow, int32_t ncol, int is_row_major, const char *parameters, const DatasetHandle reference, DatasetHandle *out)

Create dataset from dense matrix.

Return 0 when succeed, -1 when failure happens

Parameters
- data: Pointer to the data space
- data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- nrow: Number of rows
- ncol: Number of columns
- is_row_major: 1 for row-major, 0 for column-major
- parameters: Additional parameters
- reference: Used to align bin mapper with other dataset, nullptr means isn’t used
- [out] out: Created dataset

LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromMats(int32_t nmat, const void **data, int data_type, int32_t *nrow, int32_t ncol, int is_row_major, const char *parameters, const DatasetHandle reference, DatasetHandle *out)

Create dataset from array of dense matrices.

Return 0 when succeed, -1 when failure happens

Parameters
- nmat: Number of dense matrices
- data: Pointer to the data space
- data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- nrow: Number of rows
- ncol: Number of columns
- is_row_major: 1 for row-major, 0 for column-major
- parameters: Additional parameters
- reference: Used to align bin mapper with other dataset, nullptr means isn’t used
- [out] out: Created dataset
LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromSampledColumn (double ** sample_data, 
int ** sample_indices, 
int32_t ncol, const 
int * num_per_col, 
int32_t num_sample_row, 
int32_t num_total_row, 
const char * parameters, 
DatasetHandle * out)

Allocate the space for dataset and bucket feature bins according to sampled data.

Return 0 when succeed, -1 when failure happens

Parameters
• sample_data: Sampled data, grouped by the column
• sample_indices: Indices of sampled data
• ncol: Number of columns
• num_per_col: Size of each sampling column
• num_sample_row: Number of sampled rows
• num_total_row: Number of total rows
• parameters: Additional parameters
• [out] out: Created dataset

LIGHTGBM_C_EXPORT int LGBM_DatasetDumpText (DatasetHandle handle, const char * filename)

Save dataset to text file, intended for debugging use only.

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of dataset
• filename: The name of the file

LIGHTGBM_C_EXPORT int LGBM_DatasetFree (DatasetHandle handle)

Free space for dataset.

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of dataset to be freed

LIGHTGBM_C_EXPORT int LGBM_DatasetGetFeatureNames (DatasetHandle handle, const 
int len, int * num_feature_names, 
const size_t buffer_len, size_t 
*out_buffer_len, char ** feature_names)

Get feature names of dataset.

Return 0 when succeed, -1 when failure happens
Parameters

- handle: Handle of dataset
- len: Number of char* pointers stored at out_strs. If smaller than the max size, only this many strings are copied
- [out] num_feature_names: Number of feature names
- buffer_len: Size of pre-allocated strings. Content is copied up to buffer_len - 1 and null-terminated
- [out] out_buffer_len: String sizes required to do the full string copies
- [out] feature_names: Feature names, should pre-allocate memory

```c
LIGHTGBM_C_EXPORT int LGBM_DatasetGetField(DatasetHandle handle, const char *field_name, int *out_len, const void **out_ptr, int *out_type)
```

Get info vector from dataset.

**Return** 0 when succeed, -1 when failure happens

Parameters

- handle: Handle of dataset
- field_name: Field name
- [out] out_len: Used to set result length
- [out] out_ptr: Pointer to the result
- [out] out_type: Type of result pointer, can be C_API_DTYPE_INT32, C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64

```c
LIGHTGBM_C_EXPORT int LGBM_DatasetGetNumData(DatasetHandle handle, int *out)
```

Get number of data points.

**Return** 0 when succeed, -1 when failure happens

Parameters

- handle: Handle of dataset
- [out] out: The address to hold number of data points

```c
LIGHTGBM_C_EXPORT int LGBM_DatasetGetNumFeature(DatasetHandle handle, int *out)
```

Get number of features.

**Return** 0 when succeed, -1 when failure happens

Parameters

- handle: Handle of dataset
- [out] out: The address to hold number of features
LIGHTGBM_C_EXPORT int LGBM_DatasetGetSubset (const DatasetHandle handle, const int32_t *used_row_indices, int32_t num_used_row_indices, const char *parameters, DatasetHandle *out)

Create subset of a data.

Return  0 when succeed, -1 when failure happens

Parameters

• handle: Handle of full dataset
• used_row_indices: Indices used in subset
• num_used_row_indices: Length of used_row_indices
• parameters: Additional parameters
• [out] out: Subset of data

LIGHTGBM_C_EXPORT int LGBM_DatasetPushRows (DatasetHandle dataset, const void *data, int data_type, int32_t nrow, int32_t ncol, int32_t start_row)

Push data to existing dataset, if nrow + start_row == num_total_row, will call dataset->FinishLoad.

Return  0 when succeed, -1 when failure happens

Parameters

• dataset: Handle of dataset
• data: Pointer to the data space
• data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
• nrow: Number of rows
• ncol: Number of columns
• start_row: Row start index

LIGHTGBM_C_EXPORT int LGBM_DatasetPushRowsByCSR (DatasetHandle dataset, const void *indptr, int indptr_type, const int32_t *indices, const void *data, int data_type, int64_t nindptr, int64_t nelem, int64_t num_col, int64_t start_row)

Push data to existing dataset, if nrow + start_row == num_total_row, will call dataset->FinishLoad.

Return  0 when succeed, -1 when failure happens

Parameters

• dataset: Handle of dataset
• indptr: Pointer to row headers
• indptr_type: Type of indptr, can be C_API_DTYPE_INT32 or C_API_DTYPE_INT64
• indices: Pointer to column indices
• data: Pointer to the data space
• data_type: Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
• nindptr: Number of rows in the matrix + 1
• nelem: Number of nonzero elements in the matrix
• num_col: Number of columns
• start_row: Row start index

LIGHTGBM_C_EXPORT int LGBM_DatasetSaveBinary (DatasetHandle handle, const char *filename)

Save dataset to binary file.

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of dataset
• filename: The name of the file

LIGHTGBM_C_EXPORT int LGBM_DatasetSetFeatureNames (DatasetHandle handle, const char **feature_names, int num_feature_names)

Save feature names to dataset.

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of dataset
• feature_names: Feature names
• num_feature_names: Number of feature names

LIGHTGBM_C_EXPORT int LGBM_DatasetSetField (DatasetHandle handle, const char *field_name, const void *field_data, int num_element, int type)

Set vector to a content in info.

Note
• group only works for C_API_DTYPE_INT32;
• label and weight only work for C_API_DTYPE_FLOAT32;
• init_score only works for C_API_DTYPE_FLOAT64.

Return 0 when succeed, -1 when failure happens

Parameters
• handle: Handle of dataset
• field_name: Field name, can be label, weight, init_score, group
• field_data: Pointer to data vector
LightGBM, Release 3.2.0.99

- **num_element**: Number of elements in `field_data`
- **type**: Type of `field_data` pointer, can be `C_API_DTYPE_INT32`, `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`

```c
LIGHTGBM_C_EXPORT int LGBM_DatasetUpdateParamChecking(const char *old_parameters, const char *new_parameters)
```

Raises errors for attempts to update dataset parameters.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- **old_parameters**: Current dataset parameters
- **new_parameters**: New dataset parameters

```c
LIGHTGBM_C_EXPORT int LGBM_FastConfigFree(FastConfigHandle fastConfig)
```

Release FastConfig object.

**Return** 0 when it succeeds, -1 when failure happens

**Parameters**
- **fastConfig**: Handle to the FastConfig object acquired with a `*FastInit()` method.

```c
LIGHTGBM_C_EXPORT const char *LGBM_GetLastError()
```

Get string message of the last error.

**Return** Error information

```c
LIGHTGBM_C_EXPORT int LGBM_NetworkFree()
```

Finalize the network.

**Return** 0 when succeed, -1 when failure happens

```c
LIGHTGBM_C_EXPORT int LGBM_NetworkInit(const char *machines, int local_listen_port, int listen_time_out, int num_machines)
```

Initialize the network.

**Return** 0 when succeed, -1 when failure happens

**Parameters**
- **machines**: List of machines in format ‘ip1:port1,ip2:port2’
- **local_listen_port**: TCP listen port for local machines
- **listen_time_out**: Socket time-out in minutes
- **num_machines**: Total number of machines

```c
LIGHTGBM_C_EXPORT int LGBM_NetworkInitWithFunctions(int num_machines, int rank, void *reduce_scatter_ext_fun, void *allgather_ext_fun)
```

Initialize the network with external collective functions.
Return 0 when succeed, -1 when failure happens

Parameters

• num_machines: Total number of machines
• rank: Rank of local machine
• reduce_scatter_ext_fun: The external reduce-scatter function
• allgather_ext_fun: The external allgather function

```c
LIGHTGBM_C_EXPORT int LGBM_RegisterLogCallback (void (*callback)) const char*
```

Register a callback function for log redirecting.

Return 0 when succeed, -1 when failure happens

Parameters

• callback: The callback function to register

```c
inline void LGBM_SetLastError (const char *msg)
```

Set string message of the last error.

Parameters

• msg: Error message
9.1 Data Structure API

**Dataset**

```python
Dataset(data[, label, reference, weight, ...])
```
Dataset in LightGBM.

**Booster**

```python
Booster([params, train_set, model_file, ...])
```
Booster in LightGBM.

**CVBooster**

```python
CVBooster()
```
CVBooster in LightGBM.

9.1.1 lightgbm.Dataset

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

Initialize Dataset.

**Parameters**

- **data** (string, numpy array, pandas DataFrame, H2O DataTable's Frame, scipy.sparse or list of numpy arrays) – Data source of Dataset. If string, it represents the path to txt file.

- **label** (list, numpy 1-D array, pandas Series / one-column DataFrame or None, optional (default=None)) – Label of the data.

- **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, pandas Series or None, optional (default=None)) – Weight for each instance.

- **group** (list, numpy 1-D array, pandas Series or None, optional (default=None)) – Group/query data. Only used in the learning-to-rank task. \( \text{sum(group)} = n \text{ samples} \). For example, if you have a 100-document dataset with \( \text{group} = [10, 20, 40, 10, 10, 10] \), that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
**init_score** (list, numpy 1-D array, pandas Series or None, optional (default=None)) – Init score 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 unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.

**params** (dict or None, optional (default=None)) – Other parameters for Dataset.

**free_raw_data** (bool, optional (default=True)) – If True, raw data is freed after constructing inner Dataset.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong><strong>init</strong></strong>(data[, label, reference, weight, ...])</td>
<td>Initialize Dataset.</td>
</tr>
<tr>
<td><strong>add_features_from</strong>(other)</td>
<td>Add features from other Dataset to the current Dataset.</td>
</tr>
<tr>
<td><strong>construct</strong>()</td>
<td>Lazy init.</td>
</tr>
<tr>
<td><strong>create_valid</strong>(data[, label, weight, group, ...])</td>
<td>Create validation data align with current Dataset.</td>
</tr>
<tr>
<td><strong>get_data</strong>()</td>
<td>Get the raw data of the Dataset.</td>
</tr>
<tr>
<td><strong>get_feature_name</strong>()</td>
<td>Get the names of columns (features) in the Dataset.</td>
</tr>
<tr>
<td><strong>get_field</strong>(field_name)</td>
<td>Get property from the Dataset.</td>
</tr>
<tr>
<td><strong>get_group</strong>()</td>
<td>Get the group of the Dataset.</td>
</tr>
<tr>
<td><strong>get_init_score</strong>()</td>
<td>Get the initial score of the Dataset.</td>
</tr>
<tr>
<td><strong>get_label</strong>()</td>
<td>Get the label of the Dataset.</td>
</tr>
<tr>
<td><strong>get_params</strong>()</td>
<td>Get the used parameters in the Dataset.</td>
</tr>
<tr>
<td><strong>get_ref_chain</strong>([ref_limit])</td>
<td>Get a chain of Dataset objects.</td>
</tr>
<tr>
<td><strong>get_weight</strong>()</td>
<td>Get the weight of the Dataset.</td>
</tr>
<tr>
<td><strong>num_data</strong>()</td>
<td>Get the number of rows in the Dataset.</td>
</tr>
<tr>
<td><strong>num_feature</strong>()</td>
<td>Get the number of columns (features) in the Dataset.</td>
</tr>
<tr>
<td><strong>save_binary</strong>(filename)</td>
<td>Save Dataset to a binary file.</td>
</tr>
<tr>
<td><strong>set_categorical_feature</strong>(categorical_feature)</td>
<td>Set categorical features.</td>
</tr>
<tr>
<td><strong>set_feature_name</strong>(feature_name)</td>
<td>Set feature name.</td>
</tr>
<tr>
<td><strong>set_field</strong>(field_name, data)</td>
<td>Set property into the Dataset.</td>
</tr>
<tr>
<td><strong>set_group</strong>(group)</td>
<td>Set group size of Dataset (used for ranking).</td>
</tr>
<tr>
<td><strong>set_init_score</strong>(init_score)</td>
<td>Set init score of Booster to start from.</td>
</tr>
<tr>
<td><strong>set_label</strong>(label)</td>
<td>Set label of Dataset.</td>
</tr>
<tr>
<td><strong>set_weight</strong>(weight)</td>
<td>Set weight of each instance.</td>
</tr>
</tbody>
</table>
Table 2 – continued from previous page

### subset(used_indices[, params])
Get subset of current Dataset.

### add_features_from(other)
Add features from other Dataset to the current Dataset.
Both Datasets must be constructed before calling this method.

**Parameters**
- **other** *(Dataset)* — The Dataset to take features from.

**Returns**
- **self** — Dataset with the new features added.

**Return type** *Dataset*

### construct()
Lazy init.

**Returns**
- **self** — Constructed Dataset object.

**Return type** *Dataset*

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

**Parameters**
- **data** *(string, numpy array, pandas DataFrame, H2O DataTable's Frame, scipy.sparse or list of numpy arrays)* — Data source of Dataset. If string, it represents the path to txt file.
- **label** *(list, numpy 1-D array, pandas Series / one-column DataFrame or None, optional (default=None))* — Label of the data.
- **weight** *(list, numpy 1-D array, pandas Series or None, optional (default=None))* — Weight for each instance.
- **group** *(list, numpy 1-D array, pandas Series or None, optional (default=None))* — Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **init_score** *(list, numpy 1-D array, pandas Series or None, optional (default=None))* — Init score for Dataset.
- **silent** *(bool, optional (default=False))* — Whether to print messages during construction.
- **params** *(dict or None, optional (default=None))* — Other parameters for validation Dataset.

**Returns**
- **valid** — Validation Dataset with reference to self.

**Return type** *Dataset*

### get_data()
Get the raw data of the Dataset.

**Returns**
- **data** — Raw data used in the Dataset construction.

**Return type** *string, numpy array, pandas DataFrame, H2O DataTable’s Frame, scipy.sparse, list of numpy arrays or None*
get_feature_name()
Get the names of columns (features) in the Dataset.

Returns feature_names – The names of columns (features) in the Dataset.

Return type list

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/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

Return type numpy array or None

get_init_score()
Get the initial score of the Dataset.

Returns init_score – Init score of Booster.

Return type numpy array or None

get_label()
Get the label of the Dataset.

Returns label – The label information from the Dataset.

Return type numpy array or None

get_params()
Get the used parameters in the Dataset.

Returns params – The used parameters in this Dataset object.

Return type dict or None

get_ref_chain(ref_limit=100)
Get a chain of Dataset objects.

Starts with r, then goes 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 or None
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 a binary file.

Note: Please note that init_score is not saved in binary file. If you need it, please set it again after loading Dataset.

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

Returns self – Returns self.

Return type Dataset

set_categorical_feature(categorical_feature)
Set categorical features.

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

Returns self – Dataset with set categorical features.

Return type Dataset

set_feature_name(feature_name)
Set feature name.

Parameters feature_name (list of strings) – Feature names.

Returns self – Dataset with set feature name.

Return type Dataset

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

Parameters

• field_name (string) – The field name of the information.

• data (list, numpy 1-D array, pandas Series or None) – The array of data to be set.

Returns self – Dataset with set property.

Return type Dataset

set_group(group)
Set group size of Dataset (used for ranking).
Parameters group (list, numpy 1-D array, pandas Series or None) – Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

Returns self – Dataset with set group.

Return type Dataset

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

Parameters init_score (list, numpy 1-D array, pandas Series or None) – Init score for Booster.

Returns self – Dataset with set init score.

Return type Dataset

set_label(label)
Set label of Dataset.

Parameters label (list, numpy 1-D array, pandas Series / one-column DataFrame or None) – The label information to be set into Dataset.

Returns self – Dataset with set label.

Return type Dataset

set_reference(reference)
Set reference Dataset.

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

Returns self – Dataset with set reference.

Return type Dataset

set_weight(weight)
Set weight of each instance.

Parameters weight (list, numpy 1-D array, pandas Series or None) – Weight to be set for each data point.

Returns self – Dataset with set weight.

Return type Dataset

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)) – These parameters will be passed to Dataset constructor.

Returns subset – Subset of the current Dataset.

Return type Dataset
9.1.2 lightgbm.Booster

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

Booster in LightGBM.

    __init__ (params=None, train_set=None, model_file=None, model_str=None, silent=False)
        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.
    • model_str (string or None, optional (default=None)) – Model will be loaded from this string.
    • silent (bool, optional (default=False)) – Whether to print messages during construction.

Methods

    __init__ ([params, train_set, model_file, ...]) Initialize the Booster.
    add_valid (data, name) Add validation data.
    attr (key) Get attribute string from the Booster.
    current_iteration () Get the index of the current iteration.
    dump_model ([num_iteration, start_iteration, ...]) Dump Booster to JSON format.
    eval (data, name[, feval]) Evaluate for data.
    eval_train ([feval]) Evaluate for training data.
    eval_valid ([feval]) Evaluate for validation data.
    feature_importance ([importance_type, iteration]) Get feature importances.
    feature_name () Get names of features.
    free_dataset () Free Booster’s Datasets.
    free_network () Free Booster’s network.
    get_leaf_output (tree_id, leaf_id) Get the output of a leaf.
    get_split_value_histogram (feature[, bins, ...]) Get split value histogram for the specified feature.
    lower_bound () Get lower bound value of a model.
    model_from_string (model_str[, verbose]) Load Booster from a string.
    model_to_string ([num_iteration, ...]) Save Booster to string.
    num_feature () Get number of features.
    num_model_per_iteration () Get number of models per iteration.
    num_trees () Get number of weak sub-models.
    predict (data[, start_iteration, ...]) Make a prediction.
    refit (data, label[, decay_rate]) Refit the existing Booster by new data.
    reset_parameter (params) Reset parameters of Booster.

continues on next page
add_valid (data, name)
Add validation data.

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

Returns self – Booster with set validation data.
Return type Booster

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 does 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=None, start_iteration=0, importance_type='split')
Dump Booster to JSON format.

Parameters
- **num_iteration** (int or None, optional (default=None)) – Index of the iteration that should be dumped. If None, if the best iteration exists, it is dumped; otherwise, all iterations are dumped. If <= 0, all iterations are dumped.
- **start_iteration** (int, optional (default=0)) – Start index of the iteration that should be dumped.
- **importance_type** (string, optional (default="split")) – What type of feature importance should be dumped. 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 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))* – Customized evaluation function. Should accept two parameters: `preds`, `eval_data`, and return `(eval_name, eval_result, is_higher_better)` or list of such tuples.

  **preds** [list or numpy 1-D array] The predicted values.

  **eval_data** [Dataset] The evaluation dataset.

  **eval_name** [string] The name of evaluation function (without whitespaces).

  **eval_result** [float] The eval result.

  **is_higher_better** [bool] Is eval result higher better, e.g. AUC is

  For binary task, the `preds` is probability of positive class (or margin in case of specified `fobj`). For multi-class task, the `preds` is group by class_id first, then group by row_id. If you want to get i-th row `preds` in j-th class, the access way is `preds[j * num_data + i]`.

**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))* – Customized evaluation function. Should accept two parameters: `preds`, `train_data`, and return `(eval_name, eval_result, is_higher_better)` or list of such tuples.

  **preds** [list or numpy 1-D array] The predicted values.

  **train_data** [Dataset] The training dataset.

  **eval_name** [string] The name of evaluation function (without whitespaces).

  **eval_result** [float] The eval result.

  **is_higher_better** [bool] Is eval result higher better, e.g. AUC is

  For binary task, the `preds` is probability of positive class (or margin in case of specified `fobj`). For multi-class task, the `preds` is group by class_id first, then group by row_id. If you want to get i-th row `preds` in j-th class, the access way is `preds[j * num_data + i]`.

**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))* – Customized evaluation function. Should accept two parameters: `preds`, `valid_data`, and return `(eval_name, eval_result, is_higher_better)` or list of such tuples.

  **preds** [list or numpy 1-D array] The predicted values.
LightGBM, Release 3.2.0.99

valid_data  [Dataset] The validation dataset.

eval_name  [string] The name of evaluation function (without whitespaces).

eval_result  [float] The eval result.

is_higher_better  [bool] Is eval result higher better, e.g. AUC is

For binary task, the preds is probability of positive class (or margin in case of specified
fobj). For multi-class task, the preds is group by class_id first, then group by row_id. If
you want to get i-th row preds in j-th class, the access way is preds[j * num_data + i].

Returns  result – List with evaluation results.

Return type  list

feature_importance (importance_type='split', iteration=None)

Get feature importances.

Parameters

• importance_type  (string, optional (default="split").) – How
the importance is calculated. If “split”, result contains numbers of times the fea-
ture is used in a model. If “gain”, result contains total gains of splits which use the
feature.

• iteration  (int or None, optional (default=None)) – Limit num-
ber of iterations in the feature importance calculation. If None, if the best iteration
exists, it is used; otherwise, all trees are used. If <= 0, all trees are used (no limits).

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.

Returns  self – Booster without Datasets.

Return type  Booster

free_network ()

Free Booster’s network.

Returns  self – Booster with freed network.

Return type  Booster

generate_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.
get_split_value_histogram(feature, bins=None, xgboost_style=False)

Get split value histogram for the specified feature.

Parameters

- **feature** (int or string) – The feature name or index the histogram is calculated for. If int, interpreted as index. If string, interpreted as name.

**Warning:** Categorical features are not supported.

- **bins** (int, string or None, optional (default=None)) – The maximum number of bins. If None, or int and > number of unique split values and xgboost_style=True, the number of bins equals number of unique split values. If string, it should be one from the list of the supported values by numpy.histogram() function.

- **xgboost_style** (bool, optional (default=False)) – Whether the returned result should be in the same form as it is in XGBoost. If False, the returned value is tuple of 2 numpy arrays as it is in numpy.histogram() function. If True, the returned value is matrix, in which the first column is the right edges of non-empty bins and the second one is the histogram values.

Returns

- **result_tuple** (tuple of 2 numpy arrays) – If xgboost_style=False, the values of the histogram of used splitting values for the specified feature and the bin edges.

- **result_array_like** (numpy array or pandas DataFrame (if pandas is installed)) – If xgboost_style=True, the histogram of used splitting values for the specified feature.

lower_bound()

Get lower bound value of a model.

Returns **lower_bound** – Lower bound value of the model.

Return type double

model_from_string(model_str, verbose=True)

Load Booster from a string.

Parameters

- **model_str** (string) – Model will be loaded from this string.

- **verbose** (bool, optional (default=True)) – Whether to print messages while loading model.

Returns **self** – Loaded Booster object.

Return type **Booster**

model_to_string(num_iteration=None, start_iteration=0, importance_type='split')

Save Booster to string.

Parameters

- **num_iteration** (int or None, optional (default=None)) – Index of the iteration that should be saved. If None, if the best iteration exists, it is saved; otherwise, all iterations are saved. If <= 0, all iterations are saved.
• **start_iteration**(int, optional (default=0)) – Start index of the iteration that should be saved.

• **importance_type**(string, optional (default="split")) – What type of feature importance should be saved. 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** `str_repr` – String representation of Booster.

**Return type** `string`

num_feature()  
Get number of features.

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

**Return type** `int`

num_model_per_iteration()  
Get number of models per iteration.

**Returns** `model_per_iter` – The number of models per iteration.

**Return type** `int`

num_trees()  
Get number of weak sub-models.

**Returns** `num_trees` – The number of weak sub-models.

**Return type** `int`

**predict**(data, start_iteration=0, num_iteration=None, raw_score=False, pred_leaf=False, pred_contrib=False, data_has_header=False, is_reshape=True, **kwargs)  
Make a prediction.

**Parameters**

• **data**(string, numpy array, pandas DataFrame, H2O DataTable's Frame or scipy.sparse) – Data source for prediction. If string, it represents the path to txt file.

• **start_iteration**(int, optional (default=0)) – Start index of the iteration to predict. If <= 0, starts from the first iteration.

• **num_iteration**(int or None, optional (default=None)) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If <= 0, all iterations from start_iteration are used (no limits).

• **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.

**Note:** If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.

Chapter 9. Python API
Note that unlike the shap package, with `pred_contrib` we return a matrix with an extra column, where the last column is the expected value.

- **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].
- ****kwargs – Other parameters for the prediction.

Returns **result** – Prediction result. Can be sparse or a list of sparse objects (each element represents predictions for one class) for feature contributions (when `pred_contrib=True`).

Return type numpy array, scipy.sparse or list of scipy.sparse

**refit** *(data, label, decay_rate=0.9, **kwargs)*

Refit the existing Booster by new data.

Parameters

- **data** (string, numpy array, pandas DataFrame, H2O DataTable's Frame or scipy.sparse) – Data source for refit. If string, it represents the path to txt file.
- **label** (list, numpy 1-D array or pandas Series / one-column DataFrame) – Label for refit.
- **decay_rate** (float, optional (default=0.9)) – Decay rate of refit, will use `leaf_output = decay_rate * old_leaf_output + (1.0 - decay_rate) * new_leaf_output` to refit trees.
- ****kwargs – Other parameters for refit. These parameters will be passed to predict method.

Returns **result** – Refitted Booster.

Return type **Booster**

**reset_parameter** *(params)*

Reset parameters of Booster.

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

Returns **self** – Booster with new parameters.

Return type **Booster**

**rollback_one_iter** ()

Rollback one iteration.

Returns **self** – Booster with rolled back one iteration.

Return type **Booster**

**save_model** *(filename, num_iteration=None, start_iteration=0, importance_type='split')*

Save Booster to file.

Parameters

- **filename** (string) – Filename to save Booster.
• **num_iteration** *(int or None, optional (default=None)) – Index of the iteration that should be saved. If None, if the best iteration exists, it is saved; otherwise, all iterations are saved. If <= 0, all iterations are saved.*

• **start_iteration** *(int, optional (default=0)) – Start index of the iteration that should be saved.*

• **importance_type** *(string, optional (default="split")) – What type of feature importance should be saved. 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.*

Return type **Booster**

- **set_attr** (**kwargs)**
  Set attributes to the Booster.

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

  Returns self – Booster with set attributes.

  Return type **Booster**

- **set_network** *(machines, local_listen_port=12400, listen_time_out=120, num_machines=1)*
  Set the network configuration.

  Parameters

  • **machines** *(list, set or string) – Names of machines.*

  • **local_listen_port** *(int, optional (default=12400)) – TCP listen port for local machines.*

  • **listen_time_out** *(int, optional (default=120)) – Socket time-out in minutes.*

  • **num_machines** *(int, optional (default=1)) – The number of machines for distributed learning application.*

  Returns self – Booster with set network.

  Return type **Booster**

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

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

  Returns self – Booster with set training Dataset name.

  Return type **Booster**

- **shuffle_models** *(start_iteration=0, end_iteration=-1)*
  Shuffle models.

  Parameters

  • **start_iteration** *(int, optional (default=0)) – The first iteration that will be shuffled.*

  • **end_iteration** *(int, optional (default=-1)) – The last iteration that will be shuffled. If <= 0, means the last available iteration.*

  Returns self – Booster with shuffled models.
Return type  

**Booster**

trees_to_dataframe()

Parse the fitted model and return in an easy-to-read pandas DataFrame.

The returned DataFrame has the following columns.

- **tree_index**: int64, which tree a node belongs to. 0-based, so a value of 6, for example, means “this node is in the 7th tree”.
- **node_depth**: int64, how far a node is from the root of the tree. The root node has a value of 1, its direct children are 2, etc.
- **node_index**: string, unique identifier for a node.
- **left_child**: string, node_index of the child node to the left of a split. None for leaf nodes.
- **right_child**: string, node_index of the child node to the right of a split. None for leaf nodes.
- **parent_index**: string, node_index of this node’s parent. None for the root node.
- **split_feature**: string, name of the feature used for splitting. None for leaf nodes.
- **split_gain**: float64, gain from adding this split to the tree. NaN for leaf nodes.
- **threshold**: float64, value of the feature used to decide which side of the split a record will go down. NaN for leaf nodes.
- **decision_type**: string, logical operator describing how to compare a value to threshold. For example, split_feature = "Column_10", threshold = 15, decision_type = "<=" means that records where Column_10 <= 15 follow the left side of the split, otherwise follows the right side of the split. None for leaf nodes.
- **missing_direction**: string, split direction that missing values should go to. None for leaf nodes.
- **missing_type**: string, describes what types of values are treated as missing.
- **value**: float64, predicted value for this leaf node, multiplied by the learning rate.
- **weight**: float64 or int64, sum of hessian (second-order derivative of objective), summed over observations that fall in this node.
- **count**: int64, number of records in the training data that fall into this node.

**Returns**  

result – Returns a pandas DataFrame of the parsed model.

Return type  pandas DataFrame

update(train_set=None, fobj=None)

Update Booster 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. Should accept two parameters: preds, train_data, and return (grad, hess).
  
  preds  [list or numpy 1-D array] The predicted values.
  
  train_data  [Dataset] The training dataset.
grad [list or numpy 1-D array] The value of the first order derivative (gradient) for each sample point.

hess [list or numpy 1-D array] The value of the second order derivative (Hessian) for each sample point.

For binary task, the preds is probability of positive class (or margin in case of specified fobj). For multi-class task, the preds is group by class_id first, then group by row_id. If you want to get i-th row preds 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

upper_bound ()

Get upper bound value of a model.

**Returns** upper_bound – Upper bound value of the model.

**Return type** double

### 9.1.3 lightgbm.CVBooster

class lightgbm.CVBooster

Bases: object

CVBooster in LightGBM.

Auxiliary data structure to hold and redirect all boosters of cv function. This class has the same methods as Booster class. All method calls are actually performed for underlying Boosters and then all returned results are returned in a list.

**boosters**

The list of underlying fitted models.

**Type** list of Booster

**best_iteration**

The best iteration of fitted model.

**Type** int

**__init__**( )

Initialize the CVBooster.

Generally, no need to instantiate manually.

**Methods**

**__init__**( )

Initialize the CVBooster.
9.2 Training API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>train(params, train_set[, num_boost_round,...])</code></td>
<td>Perform the training with given parameters.</td>
</tr>
<tr>
<td><code>cv(params, train_set[, num_boost_round,...])</code></td>
<td>Perform the cross-validation with given parameters.</td>
</tr>
</tbody>
</table>

9.2.1 lightgbm.train

```python
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 on.
- `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 on during training.
- `valid_names (list of strings or None, optional (default=None))` – Names of valid_sets.
- `fobj (callable or None, optional (default=None))` – Customized objective function. Should accept two parameters: preds, train_data, and return (grad, hess).
  - `preds [list or numpy 1-D array]` The predicted values.
  - `train_data [Dataset]` The training dataset.
  - `grad [list or numpy 1-D array]` The value of the first order derivative (gradient) for each sample point.
  - `hess [list or numpy 1-D array]` The value of the second order derivative (Hessian) for each sample point.

For binary task, the preds is margin. For multi-class task, the preds is group by class_id first, then group by row_id. If you want to get i-th row preds 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.

- `feval (callable, list of callable functions or None, optional (default=None))` – Customized evaluation function. Each evaluation function should accept two parameters: preds, train_data, and return (eval_name, eval_result, is_higher_better) or list of such tuples.
  - `preds [list or numpy 1-D array]` The predicted values.
  - `train_data [Dataset]` The training dataset.
  - `eval_name [string]` The name of evaluation function (without whitespaces).
  - `eval_result [float]` The eval result.
  - `is_higher_better [bool]` Is eval result higher better, e.g. AUC is is_higher_better.
For binary task, the preds is probability of positive class (or margin in case of specified 
fobj). For multi-class task, the preds is group by class_id first, then group by row_id. If 
you want to get i-th row preds in j-th class, the access way is preds[j * num_data + i]. To 
ignore the default metric corresponding to the used objective, set the metric parameter 
to the string "None" in params.

- **init_model**
  
  (string, Booster 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 unordered categorical columns 
  are used. All values in categorical features should be less than int32 max value 
  (2147483647). Large values could be memory consuming. Consider using consecutive 
  integers starting from zero. All negative values in categorical features will be treated 
  as missing values. The output cannot be monotonically constrained with respect to a 
  categorical feature.

- **early_stopping_rounds**
  
  (int or None, optional (default=None)) 
  – Activates early stopping. The model will train until the validation score stops im-
  proving. Validation score needs to improve at least every early_stopping_rounds 
  round(s) to continue training. Requires at least one validation data and one metric.
  If there’s more than one, will check all of them. But the training data is ignored 
  anyway. To check only the first metric, set the first_metric_only parameter 
  to True in params. The index of iteration that has the best performance will be 
  saved in the best_iteration field if early stopping logic is enabled by setting 
  early_stopping_rounds.

- **evals_result**
  
  (dict or None, optional (default=None)) – This diction-
  ary 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 `valid_sets`, 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. When your model is very large and cause the memory error, you can try to set this param to True to avoid the model conversion performed during the internal call of `model_to_string`. 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**

### 9.2.2 lightgbm.cv

`lightgbm.cv`(*params, train_set, num_boost_round=100, 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, eval_train_metric=False, return_cvbooster=False*).

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=100)*) – Number of boosting iterations.
- **folds** (*generator or iterator of (train_idx, test_idx) tuples, scikit-learn splitter object or None, optional (default=None)*) – If generator or iterator, it should yield the train and test indices for each fold. If object, it should be one of the scikit-learn splitter classes (https://scikit-learn.org/stable/modules/classes.html#splitter-classes) and have `split` method. 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)) – Customized objective function. Should accept two parameters: preds, train_data, and return (grad, hess).

  - **preds** [list or numpy 1-D array] The predicted values.
  - **train_data** [Dataset] The training dataset.
  - **grad** [list or numpy 1-D array] The value of the first order derivative (gradient) for each sample point.
  - **hess** [list or numpy 1-D array] The value of the second order derivative (Hessian) for each sample point.

For binary task, the preds is margin. For multi-class task, the preds is group by class_id first, then group by row_id. If you want to get i-th row preds 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.

**feval** (callable, list of callable functions or None, optional (default=None)) – Customized evaluation function. Each evaluation function should accept two parameters: preds, train_data, and return (eval_name, eval_result, is_higher_better) or list of such tuples.

  - **preds** [list or numpy 1-D array] The predicted values.
  - **train_data** [Dataset] The training dataset.
  - **eval_name** [string] The name of evaluation function (without whitespaces).
  - **eval_result** [float] The eval result.
  - **is_higher_better** [bool] Is eval result higher better, e.g. AUC is is_higher_better.

For binary task, the preds is probability of positive class (or margin in case of specified fobj). For multi-class task, the preds is group by class_id first, then group by row_id. If you want to get i-th row preds in j-th class, the access way is preds[j * num_data + i]. To ignore the default metric corresponding to the used objective, set metrics to the string "None".

**init_model** (string, Booster 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 unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.
• **early_stopping_rounds** *(int or None, optional (default=None))*
  Acts as early stopping. CV score needs to improve at least every `early_stopping_rounds` round(s) to continue. Requires at least one metric. If there’s more than one, will check all of them. To check only the first metric, set the `first_metric_only` parameter to `True` in `params`. Last entry in evaluation history is the one from the 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 contain 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.

• **eval_train_metric** *(bool, optional (default=False))* – Whether to display the train metric in progress. The score of the metric is calculated again after each training step, so there is some impact on performance.

• **return_cvbooster** *(bool, optional (default=False))* – Whether to return Booster models trained on each fold through CVBooster.

Returns `eval_hist` – Evaluation history. The dictionary has the following format: `{‘metric1-mean’: [values], ‘metric1-stdv’: [values], ‘metric2-mean’: [values], ‘metric2-stdv’: [values], ...}`. If `return_cvbooster=True`, also returns trained boosters via `cvbooster` key.

**Return type** `dict`

### 9.3 Scikit-learn API

<table>
<thead>
<tr>
<th>LGBMModel([boosting_type, num_leaves, ...])</th>
<th>Implementation of the scikit-learn API for LightGBM.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGBMClassifier([boosting_type, num_leaves, ...])</td>
<td>LightGBM classifier.</td>
</tr>
<tr>
<td>LGBMRegressor([boosting_type, num_leaves, ...])</td>
<td>LightGBM regressor.</td>
</tr>
<tr>
<td>LGBMRanker([boosting_type, num_leaves, ...])</td>
<td>LightGBM ranker.</td>
</tr>
</tbody>
</table>

---

9.3. Scikit-learn API 115
9.3.1 lightgbm.LGBMModel

```python
class lightgbm.LGBMModel(booster_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', **kwargs)

Bases: object
```

Implementation of the scikit-learn API for LightGBM.

```python
__init__(booster_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', **kwargs)
```

Construct a gradient boosting model.

**Parameters**

- **num_leaves** (int, optional (default=31)) – Maximum tree leaves for base learners.
- **max_depth** (int, optional (default=-1)) – Maximum tree depth for base learners, <=0 means no limit.
- **learning_rate** (float, optional (default=0.1)) – Boosting learning rate. You can use callbacks parameter of fit method to shrink/adapt learning rate in training using resetParameter callback. Note, that this will ignore the learning_rate argument in training.
- **n_estimators** (int, optional (default=100)) – Number of boosted trees to fit.
- **subsample_for_bin** (int, optional (default=200000)) – 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: ‘regression’ for LGBMRegressor, ‘binary’ or ‘multiclass’ for LGBMClassifier, ‘lambdarank’ for LGBMRanker.
- **class_weight** (dict, 'balanced' or None, optional (default=None)) – Weights associated with classes in the form {class_label: weight}. Use this parameter only for multi-class classification task; for binary classification task you may use is_unbalance or scale_pos_weight parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (https://scikit-learn.org/stable/modules/calibration.html) of your model. The ‘balanced’ mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)). If None, all classes are supposed to have weight one. Note,
that these weights will be multiplied with sample_weight (passed through the
fit method) if sample_weight is specified.

- **min_split_gain**(float, optional (default=0.)) – Minimum loss re-
duction required to make a further partition on a leaf node of the tree.

- **min_child_weight**(float, optional (default=1e-3)) – Minimum
sum of instance weight (hessian) needed in a child (leaf).

- **min_child_samples**(int, optional (default=20)) – Minimum num-
ber of data needed in a child (leaf).

- **subsample**(float, optional (default=1.)) – Subsample ratio of the
training instance.

- **subsample_freq**(int, optional (default=0)) – Frequence of subsam-
ple, <=0 means no enable.

- **colsample_bytree**(float, optional (default=1.)) – Subsample ra-
tio 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, RandomState object or None, optional
(default=None)) – Random number seed. If int, this number is used to seed the
C++ code. If RandomState object (numpy), a random integer is picked based on its
state to seed the C++ code. If None, default seeds in C++ code are used.

- **n_jobs**(int, optional (default=-1)) – Number of parallel threads.

- **silent**(bool, optional (default=True)) – Whether to print messages
while running boosting.

- **importance_type**(string, optional (default='split')) – The
type of feature importance to be filled into feature_importances_. If 'split',
result contains numbers of times the feature is used in a model. If ‘gain’, result con-
tains total gains of splits which use the feature.

- ****kwargs – Other parameters for the model. Check http://lightgbm.readthedocs.io/
en/latest/Parameters.html for more parameters.

---

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

---

**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. Only used in the learning-to-rank task. sum(group)
  = n_samples. For example, if you have a 100-document dataset with group = [10,
  20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records
are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

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

For binary task, the y_pred is margin. 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.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Construct a gradient boosting model.</td>
</tr>
<tr>
<td>fit(X, y[, sample_weight, init_score, ...])</td>
<td>Build a gradient boosting model from the training set (X, y).</td>
</tr>
<tr>
<td>get_params([deep])</td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td>predict(X[, raw_score, start_iteration, ...])</td>
<td>Return the predicted value for each sample.</td>
</tr>
<tr>
<td>set_params(**params)</td>
<td>Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>best_iteration_</td>
<td>The best iteration of fitted model if early_stopping_rounds has been specified.</td>
</tr>
<tr>
<td>best_score_</td>
<td>The best score of fitted model.</td>
</tr>
<tr>
<td>booster_</td>
<td>The underlying Booster of this model.</td>
</tr>
<tr>
<td>evals_result_</td>
<td>The evaluation results if early_stopping_rounds has been specified.</td>
</tr>
<tr>
<td>feature_importances_</td>
<td>The feature importances (the higher, the more important).</td>
</tr>
<tr>
<td>feature_name_</td>
<td>The names of features.</td>
</tr>
<tr>
<td>n_features_</td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td>n_features_in_</td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td>objective_</td>
<td>The concrete objective used while fitting this model.</td>
</tr>
</tbody>
</table>

**property best_iteration_**

The best iteration of fitted model if early_stopping_rounds has been specified.

*Type* int or None

**property best_score_**

The best score of fitted model.

*Type* dict or None

**property booster_**

The underlying Booster of this model.

*Type* Booster
property evals_result_
The evaluation results if early_stopping_rounds has been specified.

Type dict or None

property feature_importances_
The feature importances (the higher, the more important).

Note: importance_type attribute is passed to the function to configure the type of importance values to be extracted.

Type array of shape = [n_features]

property feature_name_
The names of features.

Type array of shape = [n_features]

fit (X, y, sample_weight=None, init_score=None, eval_set=None, eval_names=None, eval_sample_weight=None, eval_class_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, init_model=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 or None, optional (default=None)) – Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

• eval_set (list or None, optional (default=None)) – A list of (X, y) tuple pairs to use as validation sets.

• 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_class_weight (list or None, optional (default=None)) – Class 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, callable, list 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 below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.

• **early_stopping_rounds** *(int or None, optional (default=None)) –* Activates early stopping. The model will train until the validation score stops improving. Validation score needs to improve at least every `early_stopping_rounds` round(s) to continue training. Requires at least one validation data and one metric. If there’s more than one, will check all of them. But the training data is ignored anyway. To check only the first metric, set the `first_metric_only` parameter to True in additional parameters **kwargs of the model constructor.

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

**Example**

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

• **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 unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.

• **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.

• **init_model** *(string, Booster, LGBMModel or None, optional (default=None)) –* Filename of LightGBM model, Booster instance or LGBMModel instance used for continue training.

**Returns**  
self – Returns self.

**Return type**  
object
**Note:** Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns `(eval_name, eval_result, is_higher_better)` or list of `(eval_name, eval_result, is_higher_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 task)] The predicted values.
- `weight` [array-like of shape = [n_samples]] The weight of samples.
- `group` [array-like] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- `eval_name` [string] The name of evaluation function (without whitespaces).
- `eval_result` [float] The eval result.
- `is_higher_better` [bool] Is eval result higher better, e.g. AUC is `is_higher_better`.

For binary task, the `y_pred` is probability of positive class (or margin in case of custom `objective`). 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].`

```python
get_params (deep=True)
Get parameters for this estimator.

Parameters deep (bool, optional (default=True)) – If True, will return the pa-
rameters for this estimator and contained subobjects that are estimators.

Returns params – Parameter names mapped to their values.

Return type dict
```

**property n_features_**
The number of features of fitted model.

**property n_features_in_**
The number of features of fitted model.

**property objective_**
The concrete objective used while fitting this model.

```python
predict (X, raw_score=False, start_iteration=0, num_iteration=None, pred_leaf=False, pred_contrib=False, **kwargs)
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.
```
• **start_iteration** (int, optional (default=0)) – Start index of the iteration to predict. If <= 0, starts from the first iteration.

• **num_iteration** (int or None, optional (default=None)) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If <= 0, all iterations from start_iteration are used (no limits).

• **pred_leaf** (bool, optional (default=False)) – Whether to predict leaf index.

• **pred_contrib** (bool, optional (default=False)) – Whether to predict feature contributions.

Note: If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with pred_contrib we return a matrix with an extra column, where the last column is the expected value.

• ****kwargs – Other parameters for the prediction.

Returns

• **predicted_result** (array-like of shape = [n_samples] or shape = [n_samples, n_classes]) – The predicted values.

• **X_leaves** (array-like of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes]) – If pred_leaf=True, the predicted leaf of every tree for each sample.

• **X_SHAP_values** (array-like of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes] or list with n_classes length of such objects) – If pred_contrib=True, the feature contributions for each sample.

**set_params**( **params**)  
Set the parameters of this estimator.

Parameters **params** – Parameter names with their new values.

Returns self – Returns self.

Return type **object**

9.3.2 lightgbm.LGBMClassifier

class lightgbm.LGBMClassifier(booster=’gbdt’, num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', **kwargs)

Bases: lightgbm.sklearn.LGBMModel, object

LightGBM classifier.
__init__(boosting_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', **kwargs)

Construct a gradient boosting model.

Parameters

- `num_leaves` (int, optional (default=31)) – Maximum tree leaves for base learners.
- `max_depth` (int, optional (default=-1)) – Maximum tree depth for base learners, <=0 means no limit.
- `learning_rate` (float, optional (default=0.1)) – Boosting learning rate. You can use callbacks parameter of fit method to shrink/adapt learning rate in training using reset parameter callback. Note, that this will ignore the learning_rate argument in training.
- `n_estimators` (int, optional (default=100)) – Number of boosted trees to fit.
- `subsample_for_bin` (int, optional (default=200000)) – 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: ‘regression’ for LGBMRegressor, ‘binary’ or ‘multiclass’ for LGBMClassifier, ‘lambdarank’ for LGBMRanker.
- `class_weight` (dict, 'balanced' or None, optional (default=None)) – Weights associated with classes in the form {class_label: weight}. Use this parameter only for multi-class classification task: for binary classification task you may use is_unbalance or scale_pos_weight parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (https://scikit-learn.org/stable/modules/calibration.html) of your model. The ‘balanced’ mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)). If None, all classes are supposed to have weight one. Note, that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.
- `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` (float, optional (default=1e-3)) – Minimum sum of instance weight (hessian) needed in a child (leaf).
- `min_child_samples` (int, optional (default=20)) – Minimum number of data needed in a child (leaf).
- `subsample` (float, optional (default=1.)) – Subsample ratio of the training instance.
• **subsample_freq**(int, optional (default=0)) – Frequency of subsample, <=0 means no enable.

• **colsample_bytree**(float, optional (default=1.0)) – 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, RandomState object or None, optional (default=None)) – Random number seed. If int, this number is used to seed the C++ code. If RandomState object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.

• **n_jobs**(int, optional (default=-1)) – Number of parallel threads.

• **silent**(bool, optional (default=True)) – Whether to print messages while running boosting.

• **importance_type**(string, optional (default='split')) – The type of feature importance to be filled into feature_importances_. 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.

• ****kwargs** – Other parameters for the model. Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

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

**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. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **grad** [array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)] The value of the first order derivative (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 order derivative (Hessian) for each sample point.

For binary task, the y_pred is margin. 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.
### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>Construct a gradient boosting model.</td>
</tr>
<tr>
<td><code>fit(X, y[, sample_weight, init_score, ...])</code></td>
<td>Build a gradient boosting model from the training set (X, y).</td>
</tr>
<tr>
<td><code>get_params([deep])</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>predict(X[, raw_score, start_iteration, ...])</code></td>
<td>Return the predicted value for each sample.</td>
</tr>
<tr>
<td><code>predict_proba(X[, raw_score, ...])</code></td>
<td>Return the predicted probability for each class for each sample.</td>
</tr>
<tr>
<td><code>set_params(**params)</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>best_iteration_</code></td>
<td>The best iteration of fitted model if <code>early_stopping_rounds</code> has been specified.</td>
</tr>
<tr>
<td><code>best_score_</code></td>
<td>The best score of fitted model.</td>
</tr>
<tr>
<td><code>booster_</code></td>
<td>The underlying Booster of this model.</td>
</tr>
<tr>
<td><code>classes_</code></td>
<td>The class label array.</td>
</tr>
<tr>
<td><code>evals_result_</code></td>
<td>The evaluation results if <code>early_stopping_rounds</code> has been specified.</td>
</tr>
<tr>
<td><code>feature_importances_</code></td>
<td>The feature importances (the higher, the more important).</td>
</tr>
<tr>
<td><code>feature_name_</code></td>
<td>The names of features.</td>
</tr>
<tr>
<td><code>n_classes_</code></td>
<td>The number of classes.</td>
</tr>
<tr>
<td><code>n_features_</code></td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td><code>n_features_in_</code></td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td><code>objective_</code></td>
<td>The concrete objective used while fitting this model.</td>
</tr>
</tbody>
</table>

**property best_iteration_**

The best iteration of fitted model if `early_stopping_rounds` has been specified.

Type: `int` or `None`

**property best_score_**

The best score of fitted model.

Type: `dict` or `None`

**property booster_**

The underlying Booster of this model.

Type: `Booster`

**property classes_**

The class label array.

Type: `array` of shape `[n_classes]`

**property evals_result_**

The evaluation results if `early_stopping_rounds` has been specified.

Type: `dict` or `None`
**property feature_importances_**

The feature importances (the higher, the more important).

**Note:** importance_type attribute is passed to the function to configure the type of importance values to be extracted.

Type array of shape = [n_features]

**property feature_name_**

The names of features.

Type array of shape = [n_features]

**fit** (X, y, sample_weight=None, init_score=None, eval_set=None, eval_names=None, eval_sample_weight=None, eval_class_weight=None, eval_init_score=None, eval_metric=None, early_stopping_rounds=None, verbose=True, feature_name='auto', categorical_feature='auto', callbacks=None, init_model=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.
- **eval_set** (list or None, optional (default=None)) – A list of (X, y) tuple pairs to use as validation sets.
- **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_class_weight** (list or None, optional (default=None)) – Class weights of eval data.
- **eval_init_score** (list of arrays or None, optional (default=None)) – Init score of eval data.
- **eval_metric** (string, callable, list 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 below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.
- **early_stopping_rounds** (int or None, optional (default=None)) – Activates early stopping. The model will train until the validation score stops improving. Validation score needs to improve at least every early_stopping_rounds round(s) to continue training. Requires at
least one validation data and one metric. If there’s more than one, will check all of them. But the training data is ignored anyway. To check only the first metric, set the `first_metric_only` parameter to True in additional parameters **kwargs of the model constructor.

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

**Example**

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

- **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 unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.

- **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.

- **init_model** (string, Booster, LGBMModel or None, optional (default=None)) – Filename of LightGBM model, Booster instance or LGBMModel instance used for continue training.

**Returns** self – Returns self.

**Return type** object

**Note:** Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns `(eval_name, eval_result, is_higher_better)` or list of `(eval_name, eval_result, is_higher_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 task)] The predicted values.
- **weight** [array-like of shape = [n_samples]] The weight of samples.
- **group** [array-like] Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
eval_name  [string] The name of evaluation function (without whitespaces).

eval_result  [float] The eval result.

is_higher_better  [bool] Is eval result higher better, e.g. AUC is is_higher_better.

For binary task, the y_pred is probability of positive class (or margin in case of custom objective).
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].

get_params (deep=True)
Get parameters for this estimator.

Parameters
depth (bool, optional (default=True)) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type dict

property n_classes_
The number of classes.

Type int

property n_features_
The number of features of fitted model.

Type int

property n_features_in_
The number of features of fitted model.

Type int

property objective_
The concrete objective used while fitting this model.

Type string or callable

predict (X, raw_score=False, start_iteration=0, num_iteration=None, pred_leaf=False, pred_contrib=False, **kwargs)
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.

• start_iteration (int, optional (default=0)) – Start index of the iteration to predict. If <= 0, starts from the first iteration.

• num_iteration (int or None, optional (default=None)) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If <= 0, all iterations from start_iteration are used (no limits).

• pred_leaf (bool, optional (default=False)) – Whether to predict leaf index.
- **pred_contrib** (bool, optional (default=False)) – Whether to predict feature contributions.

**Note:** If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with **pred_contrib** we return a matrix with an extra column, where the last column is the expected value.

- **kwargs** – Other parameters for the prediction.

**Returns**

- **predicted_result** (array-like of shape = [n_samples] or shape = [n_samples, n_classes]) – The predicted values.
- **X_leaves** (array-like of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes]) – If **pred_leaf=True**, the predicted leaf of every tree for each sample.
- **X_SHAP_values** (array-like of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes] or list with n_classes length of such objects) – If **pred_contrib=True**, the feature contributions for each sample.

**predict_proba** (**X**, **raw_score=False**, **start_iteration=0**, **num_iteration=None**, **pred_leaf=False**, **pred_contrib=False**, **kwargs**)

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.
- **start_iteration** (int, optional (default=0)) – Start index of the iteration to predict. If <= 0, starts from the first iteration.
- **num_iteration** (int or None, optional (default=None)) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If <= 0, all iterations from start_iteration are used (no limits).
- **pred_leaf** (bool, optional (default=False)) – Whether to predict leaf index.
- **pred_contrib** (bool, optional (default=False)) – Whether to predict feature contributions.

**Note:** If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with **pred_contrib** we return a matrix with an extra column, where the last column is the expected value.

- **kwargs** – Other parameters for the prediction.

**Returns**
• **predicted_probability** (array-like of shape = [n_samples] or shape = [n_samples, n_classes]) – The predicted values.

• **X_leaves** (array-like of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes]) – If pred_leaf=True, the predicted leaf of every tree for each sample.

• **X_SHAP_values** (array-like of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes] or list with n_classes length of such objects) – If pred_contrib=True, the feature contributions for each sample.

```python
set_params(**params)
```

Set the parameters of this estimator.

Parameters

**params** – Parameter names with their new values.

Returns

self – Returns self.

Return type

object

### 9.3.3 lightgbm.LGBMRegressor

class lightgbm.LGBMRegressor(booster_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.0, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', **kwargs)

Bases: lightgbm.sklearn.LGBMModel, object

LightGBM regressor.

```python
__init__(booster_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.0, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', **kwargs)
```

Construct a gradient boosting model.

Parameters


• **num_leaves**(int, optional (default=31)) – Maximum tree leaves for base learners.

• **max_depth**(int, optional (default=-1)) – Maximum tree depth for base learners, <=0 means no limit.

• **learning_rate**(float, optional (default=0.1)) – Boosting learning rate. You can use callbacks parameter of fit method to shrink/adapt learning rate in training using reset_parameter callback. Note, that this will ignore the learning_rate argument in training.

• **n_estimators**(int, optional (default=100)) – Number of boosted trees to fit.
• **subsample_for_bin** *(int, optional (default=200000)) – 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: ‘regression’ for LGBMRegressor, ‘binary’ or ‘multiclass’ for LGBMClassifier, ‘lambdarank’ for LGBMRanker.*

• **class_weight** *(dict, 'balanced' or None, optional (default=None)) – Weights associated with classes in the form {class_label: weight}. Use this parameter only for multi-class classification task: for binary classification task you may use is_unbalance or scale_pos_weight parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (https://scikit-learn.org/stable/modules/calibration.html) of your model. The ‘balanced’ mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)). If None, all classes are supposed to have weight one. Note, that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

• **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** *(float, optional (default=1e-3)) – Minimum sum of instance weight (hessian) needed in a child (leaf).*

• **min_child_samples** *(int, optional (default=20)) – Minimum number of data needed in a child (leaf).*

• **subsample** *(float, optional (default=1.)) – Subsample ratio of the training instance.*

• **subsample_freq** *(int, optional (default=0)) – Frequency 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, RandomState object or None, optional (default=None)) – Random number seed. If int, this number is used to seed the C++ code. If RandomState object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.

• **n_jobs** *(int, optional (default=-1)) – Number of parallel threads.*

• **silent** *(bool, optional (default=True)) – Whether to print messages while running boosting.*

• **importance_type** *(string, optional (default='split')) – The type of feature importance to be filled into feature_importances_. 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.
• **kwargs – Other parameters for the model. Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

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

**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. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- `grad` [array-like of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task)] The value of the first order derivative (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 order derivative (Hessian) for each sample point.

For binary task, the `y_pred` is margin. 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.

**Methods**

- `__init__`([boosting_type, num_leaves, ...]) Construct a gradient boosting model.
- `fit`(X, y[, sample_weight, init_score, ...]) Build a gradient boosting model from the training set (X, y).
- `get_params`([deep]) Get parameters for this estimator.
- `predict`(X[, raw_score, start_iteration, ...]) Return the predicted value for each sample.
- `set_params`(**params) Set the parameters of this estimator.

**Attributes**

- `best_iteration_` The best iteration of fitted model if `early_stopping_rounds` has been specified.
- `best_score_` The best score of fitted model.
- `booster_` The underlying Booster of this model.
- `evals_result_` The evaluation results if `early_stopping_rounds` has been specified.
Table 12 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>feature_importances_</code></td>
<td>The feature importances (the higher, the more important).</td>
</tr>
<tr>
<td><code>feature_name_</code></td>
<td>The names of features.</td>
</tr>
<tr>
<td><code>n_features_</code></td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td><code>n_features_in_</code></td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td><code>objective_</code></td>
<td>The concrete objective used while fitting this model.</td>
</tr>
</tbody>
</table>

**property best_iteration_**
The best iteration of fitted model if `early_stopping_rounds` has been specified.

- **Type**: int or None

**property best_score_**
The best score of fitted model.

- **Type**: dict or None

**property booster_**
The underlying Booster of this model.

- **Type**: Booster

**property evals_result_**
The evaluation results if `early_stopping_rounds` has been specified.

- **Type**: dict or None

**property feature_importances_**
The feature importances (the higher, the more important).

- **Note**: `importance_type` attribute is passed to the function to configure the type of importance values to be extracted.

- **Type**: array of shape = `[n_features]`

**property feature_name_**
The names of features.

- **Type**: array of shape = `[n_features]`

**fit**
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.
• **eval_set** (list or None, optional (default=None)) – A list of (X, y) tuple pairs to use as validation sets.

• **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_metric** (string, callable, list 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 below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.

• **early_stopping_rounds** (int or None, optional (default=None)) – Activates early stopping. The model will train until the validation score stops improving. Validation score needs to improve at least every early_stopping_rounds round(s) to continue training. Requires at least one validation data and one metric. If there’s more than one, will check all of them. But the training data is ignored anyway. To check only the first metric, set the first_metric_only parameter to True in additional parameters **kwargs of the model constructor.

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

**Example**

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

• **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 unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.

• **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.
• **init_model** *(string, Booster, LGBMModel or None, optional (default=None)) –* Filename of LightGBM model, Booster instance or LGBMModel instance used for continue training.

Returns **self** – Returns self.

Return type **object**

**Note:** Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns `(eval_name, eval_result, is_higher_better)` or list of `(eval_name, eval_result, is_higher_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 task)] The predicted values.
- **weight** [array-like of shape = [n_samples]] The weight of samples.
- **group** [array-like] Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **eval_name** [string] The name of evaluation function (without whitespaces).
- **eval_result** [float] The eval result.
- **is_higher_better** [bool] Is eval result higher better, e.g. AUC is **is_higher_better**.

For binary task, the y_pred is probability of positive class (or margin in case of custom **objective**). 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].

```python
get_params(deep=True)
```
Get parameters for this estimator.

Parameters **deep** *(bool, optional (default=True)) –* If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type **dict**

**property n_features_**
The number of features of fitted model.

Type **int**

**property n_features_in_**
The number of features of fitted model.

Type **int**

**property objective_**
The concrete objective used while fitting this model.

Type **string** or **callable**

```python
predict(X, raw_score=False, start_iteration=0, num_iteration=None, pred_leaf=False, pred_contrib=False, **kwargs)
```
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.
- **start_iteration** (int, optional (default=0)) – Start index of the iteration to predict. If <= 0, starts from the first iteration.
- **num_iteration** (int or None, optional (default=None)) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If <= 0, all iterations from start_iteration are used (no limits).
- **pred_leaf** (bool, optional (default=False)) – Whether to predict leaf index.
- **pred_contrib** (bool, optional (default=False)) – Whether to predict feature contributions.

**Note:** If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with pred_contrib we return a matrix with an extra column, where the last column is the expected value.

- ****kwargs – Other parameters for the prediction.

Returns

- **predicted_result** (array-like of shape = [n_samples] or shape = [n_samples, n_classes]) – The predicted values.
- **X_leaves** (array-like of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes]) – If pred_leaf=True, the predicted leaf of every tree for each sample.
- **X_SHAP_values** (array-like of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes] or list with n_classes length of such objects) – If pred_contrib=True, the feature contributions for each sample.

**set_params**( **params**)

Set the parameters of this estimator.

Parameters **params** – Parameter names with their new values.

Returns self – Returns self.

Return type object
9.3.4 lightgbm.LGBMRanker

class lightgbm.LGBMRanker

boosting_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', **kwargs)

Bases: lightgbm.sklearn.LGBMModel

LightGBM ranker.

__init__

boosting_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', **kwargs)

Construct a gradient boosting model.

Parameters


• num_leaves (int, optional (default=31)) – Maximum tree leaves for base learners.

• max_depth (int, optional (default=-1)) – Maximum tree depth for base learners, \(<0\) means no limit.

• learning_rate (float, optional (default=0.1)) – Boosting learning rate. You can use callbacks parameter of fit method to shrink/adapt learning rate in training using reset_parameter callback. Note, that this will ignore the learning_rate argument in training.

• n_estimators (int, optional (default=100)) – Number of boosted trees to fit.

• subsample_for_bin (int, optional (default=200000)) – 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: ‘regression’ for LGBMRegressor, ‘binary’ or ‘multiclass’ for LGBMClassifier, ‘lambdarank’ for LGBMRanker.

• class_weight (dict, ‘balanced’ or None, optional (default=None)) – Weights associated with classes in the form {class_label: weight}. Use this parameter only for multi-class classification task; for binary classification task you may use is_unbalance or scale_pos_weight parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (https://scikit-learn.org/stable/modules/calibration.html) of your model. The ‘balanced’ mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)). If None, all classes are supposed to have weight one. Note,
that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

- **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**(float, optional (default=1e-3)) – Minimum sum of instance weight (hessian) needed in a child (leaf).

- **min_child_samples**(int, optional (default=20)) – Minimum number of data needed in a child (leaf).

- **subsample**(float, optional (default=1.)) – Subsample ratio of the training instance.

- **subsample_freq**(int, optional (default=0)) – Frequency 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, RandomState object or None, optional (default=None)) – Random number seed. If int, this number is used to seed the C++ code. If RandomState object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.

- **n_jobs**(int, optional (default=-1)) – Number of parallel threads.

- **silent**(bool, optional (default=True)) – Whether to print messages while running boosting.

- **importance_type**(string, optional (default='split')) – The type of feature importance to be filled into feature_importances_. 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.

- ****kwargs – Other parameters for the model. Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

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

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. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records
are in the first group, records 11-30 are in the second group, records 31-70 are in the third
group, etc.

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

For binary task, the y_pred is margin. 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.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong><strong>init</strong></strong></td>
<td>([boosting_type, num_leaves, ...]) Construct a gradient boosting model.</td>
</tr>
<tr>
<td><strong>fit</strong></td>
<td>(X, y[, sample_weight, init_score, ...]) Build a gradient boosting model</td>
</tr>
<tr>
<td></td>
<td>from the training set (X, y).</td>
</tr>
<tr>
<td><strong>get_params</strong></td>
<td>([deep]) Get parameters for this estimator.</td>
</tr>
<tr>
<td><strong>predict</strong></td>
<td>(X[, raw_score, start_iteration, ...]) Return the predicted value for each</td>
</tr>
<tr>
<td></td>
<td>sample.</td>
</tr>
<tr>
<td><strong>set_params</strong></td>
<td>(**params) Set the parameters of this estimator.</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>best_iteration_</strong></td>
<td>The best iteration of fitted model if early_stopping_rounds has been specified.</td>
</tr>
<tr>
<td><strong>best_score_</strong></td>
<td>The best score of fitted model.</td>
</tr>
<tr>
<td><strong>booster_</strong></td>
<td>The underlying Booster of this model.</td>
</tr>
<tr>
<td><strong>evals_result_</strong></td>
<td>The evaluation results if early_stopping_rounds has been specified.</td>
</tr>
<tr>
<td><strong>feature_importances_</strong></td>
<td>The feature importances (the higher, the more important).</td>
</tr>
<tr>
<td><strong>feature_name_</strong></td>
<td>The names of features.</td>
</tr>
<tr>
<td><strong>n_features_</strong></td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td><strong>n_features_in_</strong></td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td><strong>objective_</strong></td>
<td>The concrete objective used while fitting this model.</td>
</tr>
</tbody>
</table>

**property best_iteration_**

The best iteration of fitted model if early_stopping_rounds has been specified.

Type `int` or `None`

**property best_score_**

The best score of fitted model.

Type `dict` or `None`

**property booster_**

The underlying Booster of this model.

Type `Booster`
**property evals_result**

The evaluation results if `early_stopping_rounds` has been specified.

**Type** `dict` or `None`

**property feature_importances**

The feature importances (the higher, the more important).

**Note:** `importance_type` attribute is passed to the function to configure the type of importance values to be extracted.

**Type** `array` of shape = `[n_features]`

**property feature_name**

The names of features.

**Type** `array` of shape = `[n_features]`

**fit**

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 or `None`, optional (default=None)) – Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **eval_set** (list or `None`, optional (default=None)) – A list of `(X, y)` tuple pairs to use as validation sets.
- **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, callable, list 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 below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.

**eval_at**  
(iterable of int, optional (default=(1, 2, 3, 4, 5))) – The evaluation positions of the specified metric. 
early_stopping_rounds : int or None, optional (default=None) Activates early stopping. The model will train until the validation score stops improving. Validation score needs to improve at least every early_stopping_rounds round(s) to continue training. Requires at least one validation data and one metric. If there’s more than one, will check all of them. But the training data is ignored anyway. To check only the first metric, set the first_metric_only parameter to True in additional parameters **kwargs of the model constructor.

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

**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 unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.

**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.

**init_model**  
(string, Booster, LGBMModel or None, optional (default=None)) – Filename of LightGBM model, Booster instance or LGBMModel instance used for continue training.

Returns  
self – Returns self.

Return type  
object

**Note:** Custom eval function expects a callable with following signatures: func(y_true, y_pred).
func(y_true, y_pred, weight) or func(y_true, y_pred, weight, group) and returns (eval_name, eval_result, is_higher_better) or list of (eval_name, eval_result, is_higher_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 task)] The predicted values.
- **weight** [array-like of shape = [n_samples]] The weight of samples.
- **group** [array-like] Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **eval_name** [string] The name of evaluation function (without whitespaces).
- **eval_result** [float] The eval result.
- **is_higher_better** [bool] Is eval result higher better, e.g. AUC is is_higher_better.

For binary task, the y_pred is probability of positive class (or margin in case of custom objective). 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].

**get_params**(deep=True)
Get parameters for this estimator.

- **deep** (bool, optional (default=True)) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns** params – Parameter names mapped to their values.

**Return type** dict

**property** n_features_

The number of features of fitted model.

**Type** int

**property** n_features_in_

The number of features of fitted model.

**Type** int

**property** objective_

The concrete objective used while fitting this model.

**Type** string or callable

**predict** (X, raw_score=False, start_iteration=0, num_iteration=None, pred_leaf=False, pred_contrib=False, **kwargs)
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.
  - **start_iteration** (int, optional (default=0)) – Start index of the iteration to predict. If <= 0, starts from the first iteration.
• **num_iteration** *(int or None, optional (default=None)) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If <= 0, all iterations from start_iteration are used (no limits).*

• **pred_leaf** *(bool, optional (default=False)) – Whether to predict leaf index.*

• **pred_contrib** *(bool, optional (default=False)) – Whether to predict feature contributions.*

---

**Note:** If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with **pred_contrib** we return a matrix with an extra column, where the last column is the expected value.

• ****kwargs – Other parameters for the prediction.

**Returns**

• **predicted_result** *(array-like of shape = [n_samples] or shape = [n_samples, n_classes]) – The predicted values.*

• **X_leaves** *(array-like of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes]) – If **pred_leaf=True**, the predicted leaf of every tree for each sample.*

• **X_SHAP_values** *(array-like of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes] or list with n_classes length of such objects) – If **pred_contrib=True**, the feature contributions for each sample.*

---

**set_params(**params)**

Set the parameters of this estimator.

**Parameters**  
**params** – Parameter names with their new values.

**Returns**  
**self** – Returns self.

**Return type**  
**object**

---

**9.4 Dask API**

New in version 3.2.0.

**DaskLGBMClassifier((boosting_type, ...))**  
Distributed version of lightgbm.LGBMClassifier.

**DaskLGBMRegressor((boosting_type, ...))**  
Distributed version of lightgbm.LGBMRegressor.

**DaskLGBMRanker((boosting_type, num_leaves, ...))**  
Distributed version of lightgbm.LGBMRanker.
9.4.1 lightgbm.DaskLGBMClassifier

class lightgbm.DaskLGBMClassifier(booster_type='gbdt', num_leaves=31, max_depth=-1, 
learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, 
min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, 
colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, 
silent=True, importance_type='split', client=None, 
**kwargs)

Bases: lightgbm.sklearn.LGBMClassifier, lightgbm.dask._DaskLGBMModel

Distributed version of lightgbm.LGBMClassifier.

__init__ (booster_type='gbdt', num_leaves=31, max_depth=-1, 
learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, 
min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, 
colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, 
silent=True, importance_type='split', client=None, 
**kwargs)

Construct a gradient boosting model.

Parameters

- **num_leaves** (int, optional (default=31)) – Maximum tree leaves for base learners.
- **max_depth** (int, optional (default=-1)) – Maximum tree depth for base learners, <=0 means no limit.
- **learning_rate**: (float, optional (default=0.1)) – Boosting learning rate. You can use callbacks parameter of fit method to shrink/adapt learning rate in training using reset_parameter callback. Note, that this will ignore the learning_rate argument in training.
- **n_estimators**: (int, optional (default=100)) – Number of boosted trees to fit.
- **subsample_for_bin** (int, optional (default=200000)) – 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: ‘regression’ for LGBMRegressor, ‘binary’ or ‘multiclass’ for LGBMClassifier, ‘lambdarank’ for LGBMRanker.
- **class_weight** (dict, 'balanced' or None, optional (default=None)) – Weights associated with classes in the form {class_label: weight}. Use this parameter only for multi-class classification task; for binary classification task you may use is_unbalance or scale_pos_weight parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (https://scikit-learn.org/stable/modules/calibration.html) of your model. The
‘balanced’ mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as \( \frac{n_{\text{samples}}}{(n_{\text{classes}} \times \text{np.bincount}(y))} \). If None, all classes are supposed to have weight one. Note, that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

- **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** *(float, optional (default=1e-3)) – Minimum sum of instance weight (hessian) needed in a child (leaf).*

- **min_child_samples** *(int, optional (default=20)) – Minimum number of data needed in a child (leaf).*

- **subsample** *(float, optional (default=1.)) – Subsample ratio of the training instance.*

- **subsample_freq** *(int, optional (default=0)) – Frequency 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, RandomState object or None, optional (default=None)) – Random number seed. If int, this number is used to seed the C++ code. If RandomState object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.*

- **n_jobs** *(int, optional (default=-1)) – Number of parallel threads.*

- **silent** *(bool, optional (default=True)) – Whether to print messages while running boosting.*

- **importance_type** *(string, optional (default='split')) – The type of feature importance to be filled into feature_importances_. 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.*

- **client** *(dask.distributed.Client or None, optional (default=None)) – Dask client. If None, distributed.default_client() will be used at runtime. The Dask client used by this class will not be saved if the model object is pickled.*

- ****kwargs** – Other parameters for the model. Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

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

__init__((boosting_type, num_leaves, ...)) Construct a gradient boosting model.

fit(X, y[, sample_weight, init_score]) Build a gradient boosting model from the training set (X, y).

get_params([deep]) Get parameters for this estimator.

predict(X, **kwargs) Return the predicted value for each sample.

predict_proba(X, **kwargs) Return the predicted probability for each class for each sample.

set_params(**params) Set the parameters of this estimator.

to_local() Create regular version of lightgbm.LGBMClassifier from the distributed version.

Attributes

best_iteration_ The best iteration of fitted model if early_stopping_rounds has been specified.

best_score_ The best score of fitted model.

booster_ The underlying Booster of this model.

classes_ The class label array.

client_ Dask client.

evals_result_ The evaluation results if early_stopping_rounds has been specified.

feature_importances_ The feature importances (the higher, the more important).

feature_name_ The names of features.

n_classes_ The number of classes.

n_features_ The number of features of fitted model.

n_features_in_ The number of features of fitted model.

objective_ The concrete objective used while fitting this model.

property best_iteration_ The best iteration of fitted model if early_stopping_rounds has been specified.

    Type int or None

property best_score_ The best score of fitted model.

    Type dict or None

property booster_ The underlying Booster of this model.

    Type Booster

property classes_ The class label array.

    Type array of shape = [n_classes]

property client_ Dask client.
This property can be passed in the constructor or updated with `model.set_params(client=client)`.

Type: `dask.distributed.Client`

**property `evals_result_`**
The evaluation results if `early_stopping_rounds` has been specified.

Type: `dict` or `None`

**property `feature_importances_`**
The feature importances (the higher, the more important).

Note: `importance_type` attribute is passed to the function to configure the type of importance values to be extracted.

Type: `array` of shape = `[n_features]`

**property `feature_name_`**
The names of features.

Type: `array` of shape = `[n_features]`

**`fit(X, y, sample_weight=None, init_score=None, **kwargs)`**
Build a gradient boosting model from the training set (X, y).

Parameters

- **X** *(Dask Array or Dask DataFrame of shape = `[n_samples, n_features]`)* – Input feature matrix.
- **y** *(Dask Array, Dask DataFrame or Dask Series of shape = `[n_samples]`)* – The target values (class labels in classification, real numbers in regression).
- **sample_weight** *(Dask Array, Dask DataFrame, Dask Series of shape = `[n_samples]` or None, optional (default=None))* – Weights of training data.
- **init_score** *(Dask Array, Dask DataFrame, Dask Series of shape = `[n_samples]` or None, optional (default=None))* – Init score of training data.
- **verbose** *(bool or int, optional (default=True))* – Requires at least one evaluation data. If True, the eval metric on the eval set is printed at each boosting stage. If int, the eval metric on the eval set is printed at every verbose boosting stage. The last boosting stage or the boosting stage found by using `early_stopping_rounds` is also printed.
Example

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

- **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 unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.

- ****kwargs – Other parameters passed through to `LGBMClassifier.fit()`.

`get_params` *(deep=True)*

Get parameters for this estimator.

Parameters `deep` *(bool, optional (default=True))* – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns `params` – Parameter names mapped to their values.

Return type `dict`

**property n_classes_**

The number of classes.

Type `int`

**property n_features_**

The number of features of fitted model.

Type `int`

**property n_features_in_**

The number of features of fitted model.

Type `int`

**property objective_**

The concrete objective used while fitting this model.

Type `string or callable`

`predict` *(X, **kwargs)*

Return the predicted value for each sample.

Parameters

- **X** *(Dask Array or Dask DataFrame of shape = [n_samples, n_features]*) – Input features matrix.

- **raw_score** *(bool, optional (default=False))* – Whether to predict raw scores.

- **start_iteration** *(int, optional (default=0))* – Start index of the iteration to predict. If <= 0, starts from the first iteration.
- `num_iteration (int or None, optional (default=None))` – Total number of iterations used in the prediction. If None, if the best iteration exists and `start_iteration <= 0`, the best iteration is used; otherwise, all iterations from `start_iteration` are used (no limits). If `<= 0`, all iterations from `start_iteration` are used (no limits).
- `pred_leaf (bool, optional (default=False))` – Whether to predict leaf index.
- `pred_contrib (bool, optional (default=False))` – Whether to predict feature contributions.

**Note:** If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with `pred_contrib` we return a matrix with an extra column, where the last column is the expected value.

- `**kwargs` – Other parameters for the prediction.

Returns

- `predicted_result (Dask Array of shape = [n_samples] or shape = [n_samples, n_classes])` – The predicted values.
- `X_leaves (Dask Array of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes])` – If `pred_leaf=True`, the predicted leaf of every tree for each sample.
- `X_SHAP_values (Dask Array of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes])` – If `pred_contrib=True`, the feature contributions for each sample.

`predict_proba (X, **kwargs)`

Return the predicted probability for each class for each sample.

Parameters

- `X` (Dask Array or Dask DataFrame of shape = [n_samples, n_features]) – Input features matrix.
- `raw_score (bool, optional (default=False))` – Whether to predict raw scores.
- `start_iteration (int, optional (default=0))` – Start index of the iteration to predict. If `<= 0`, starts from the first iteration.
- `num_iteration (int or None, optional (default=None))` – Total number of iterations used in the prediction. If None, if the best iteration exists and `start_iteration <= 0`, the best iteration is used; otherwise, all iterations from `start_iteration` are used (no limits). If `<= 0`, all iterations from `start_iteration` are used (no limits).
- `pred_leaf (bool, optional (default=False))` – Whether to predict leaf index.
- `pred_contrib (bool, optional (default=False))` – Whether to predict feature contributions.

**Note:** If you want to get more explanations for your model’s predictions using SHAP
values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with `pred_contrib` we return a matrix with an extra column, where the last column is the expected value.

- **kwargs – Other parameters for the prediction.

Returns

- predicted_probability (Dask Array of shape = [n_samples] or shape = [n_samples, n_classes]) – The predicted values.
- X_leaves (Dask Array of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes]) – If `pred_leaf=True`, the predicted leaf of every tree for each sample.
- X_SHAP_values (Dask Array of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes]) – If `pred_contrib=True`, the feature contributions for each sample.

**set_params** (**params**)

Set the parameters of this estimator.

Parameters **params** – Parameter names with their new values.

Returns self – Returns self.

Return type object

to_local()

Create regular version of lightgbm.LGBMClassifier from the distributed version.

Returns model – Local underlying model.

Return type lightgbm.LGBMClassifier

### 9.4.2 lightgbm.DaskLGBMRegressor

**class** lightgbm.DaskLGBMRegressor (boosting_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', client=None, **kwargs)

Bases: lightgbm.sklearn.LGBMRegressor, lightgbm.dask._DaskLGBMModel

Distributed version of lightgbm.LGBMRegressor.

**init** (boosting_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', client=None, **kwargs)

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, <=0 means no limit.

• **learning_rate** *(float, optional (default=0.1))* — Boosting learning rate. You can use callbacks parameter of fit method to shrink/adapt learning rate in training using reset_parameter callback. Note, that this will ignore the learning_rate argument in training.

• **n_estimators** *(int, optional (default=100))* — Number of boosted trees to fit.

• **subsample_for_bin** *(int, optional (default=200000))* — 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: ‘regression’ for LGBMRegressor, ‘binary’ or ‘multiclass’ for LGBMClassifier, ‘lambdarank’ for LGBMRanker.

• **class_weight** *(dict, 'balanced' or None, optional (default=None)* — Weights associated with classes in the form `{class_label: weight}`. Use this parameter only for multi-class classification task; for binary classification task you may use is_unbalance or scale_pos_weight parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration ([https://scikit-learn.org/stable/modules/calibration.html](https://scikit-learn.org/stable/modules/calibration.html)) of your model. The ‘balanced’ mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)). If None, all classes are supposed to have weight one. Note, that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

• **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** *(float, optional (default=1e-3))* — Minimum sum of instance weight (hessian) needed in a child (leaf).

• **min_child_samples** *(int, optional (default=20))* — Minimum number of data needed in a child (leaf).

• **subsample** *(float, optional (default=1.))* — Subsample ratio of the training instance.

• **subsample_freq** *(int, optional (default=0))* — Frequency 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.
LightGBM, Release 3.2.0.99

- **reg_lambda** *(float, optional (default=0.)) - L2 regularization term on weights.*

- **random_state** *(int, RandomState object or None, optional (default=None)) - Random number seed. If int, this number is used to seed the C++ code. If RandomState object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.*

- **n_jobs** *(int, optional (default=-1)) - Number of parallel threads.*

- **silent** *(bool, optional (default=True)) - Whether to print messages while running boosting.*

- **importance_type** *(string, optional (default='split')) - The type of feature importance to be filled into feature_importances_. 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.*

- **client** *(dask.distributed.Client or None, optional (default=None)) - Dask client. If None, distributed.default_client() will be used at runtime. The Dask client used by this class will not be saved if the model object is pickled.*

- ****kwargs** - Other parameters for the model. Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

---

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

### Methods

- **__init__**(*([boosting_type, num_leaves, ...])*) - Construct a gradient boosting model.

- **fit**(X, y[, sample_weight, init_score]) - Build a gradient boosting model from the training set (X, y).

- **get_params**(deep) - Get parameters for this estimator.

- **predict**(X, **kwargs) - Return the predicted value for each sample.

- **set_params**(**params) - Set the parameters of this estimator.

- **to_local**() - Create regular version of lightgbm.LGBMRegressor from the distributed version.

### Attributes

- **best_iteration_** - The best iteration of fitted model if early_stopping_rounds has been specified.

- **best_score_** - The best score of fitted model.

- **booster_** - The underlying Booster of this model.

- **client_** - Dask client.

- **evals_result_** - The evaluation results if early_stopping_rounds has been specified.

---

continues on next page
Table 19 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_importances_</td>
<td>The feature importances (the higher, the more important).</td>
</tr>
<tr>
<td>feature_name_</td>
<td>The names of features.</td>
</tr>
<tr>
<td>n_features_</td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td>n_features_in_</td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td>objective_</td>
<td>The concrete objective used while fitting this model.</td>
</tr>
<tr>
<td>best_iteration_</td>
<td>The best iteration of fitted model if early_stopping_rounds has been specified.</td>
</tr>
<tr>
<td>best_score_</td>
<td>The best score of fitted model.</td>
</tr>
<tr>
<td>booster_</td>
<td>The underlying Booster of this model.</td>
</tr>
<tr>
<td>client_</td>
<td>Dask client. This property can be passed in the constructor or updated with model. set_params(client=client).</td>
</tr>
<tr>
<td>evals_result_</td>
<td>The evaluation results if early_stopping_rounds has been specified.</td>
</tr>
<tr>
<td>feature_importances_</td>
<td>The feature importances (the higher, the more important).</td>
</tr>
</tbody>
</table>

Note: importance_type attribute is passed to the function to configure the type of importance values to be extracted.

Type array of shape = [n_features]

property feature_name_         | The names of features.                                                      |
|                               | Type array of shape = [n_features]                                          |

fit (X, y, sample_weight=None, init_score=None, **kwargs) Build a gradient boosting model from the training set (X, y).

Parameters

- **X** (Dask Array or Dask DataFrame of shape = [n_samples, n_features]) – Input feature matrix.
- **y** (Dask Array, Dask DataFrame or Dask Series of shape = [n_samples]) – The target values (class labels in classification, real numbers in regression).
• **sample_weight** *(Dask Array, Dask DataFrame, Dask Series of shape = \([n\_samples]\) or None, optional (default=None)) – Weights of training data.*

• **init_score** *(Dask Array, Dask DataFrame, Dask Series of shape = \([n\_samples]\) or None, optional (default=None)) – Init score of training data.*

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

**Example**

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

• **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 unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.*

• ****kwargs** – Other parameters passed through to LGBMRegressor.fit().

**get_params** *(deep=True)*

Get parameters for this estimator.

Parameters **deep** *(bool, optional (default=True)) – If True, will return the parameters for this estimator and contained subobjects that are estimators.*

Returns **params** – Parameter names mapped to their values.

Return type **dict**

**property n_features_**

The number of features of fitted model.

Type int

**property n_features_in_**

The number of features of fitted model.

Type int

**property objective_**

The concrete objective used while fitting this model.

Type string or callable
**predict** (*X, **kwargs*)

Return the predicted value for each sample.

**Parameters**

- **X** (Dask Array or Dask DataFrame of shape = [n_samples, n_features]) – Input features matrix.
- **raw_score** (bool, optional (default=False)) – Whether to predict raw scores.
- **start_iteration** (int, optional (default=0)) – Start index of the iteration to predict. If <= 0, starts from the first iteration.
- **num_iteration** (int or None, optional (default=None)) – Total number of iterations used in the prediction. If None, if the best iteration exists and start iteration <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If <= 0, all iterations from start_iteration are used (no limits).
- **pred_leaf** (bool, optional (default=False)) – Whether to predict leaf index.
- **pred_contrib** (bool, optional (default=False)) – Whether to predict feature contributions.

**Note:** If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with **pred_contrib** we return a matrix with an extra column, where the last column is the expected value.

- ****kwargs – Other parameters for the prediction.

**Returns**

- **predicted_result** (Dask Array of shape = [n_samples]) – The predicted values.
- **X_leaves** (Dask Array of shape = [n_samples, n_trees]) – If pred_leaf=True, the predicted leaf of every tree for each sample.
- **X_SHAP_values** (Dask Array of shape = [n_samples, n_features + 1]) – If pred_contrib=True, the feature contributions for each sample.

**set_params** (**params**)

Set the parameters of this estimator.

**Parameters** **params** – Parameter names with their new values.

**Returns** self – Returns self.

**Return type** object

**to_local** ()

Create regular version of lightgbm.LGBMRegressor from the distributed version.

**Returns** model – Local underlying model.

**Return type** lightgbm.LGBMRegressor
9.4.3 lightgbm.DaskLGBMRanker

class lightgbm.DaskLGBMRanker(booster_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', client=None, **kwargs)

Bases: lightgbm.sklearn.LGBMRanker, lightgbm.dask._DaskLGBMModel

Distributed version of lightgbm.LGBMRanker.

__init__ (booster_type='gbdt', num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100, subsample_for_bin=200000, objective=None, class_weight=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20, subsample=1.0, subsample_freq=0, colsample_bytree=1.0, reg_alpha=0.0, reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, importance_type='split', client=None, **kwargs)

Construct a gradient boosting model.

Parameters


- **num_leaves** (int, optional (default=31)) – Maximum tree leaves for base learners.

- **max_depth** (int, optional (default=-1)) – Maximum tree depth for base learners, <=0 means no limit.

- **learning_rate** (float, optional (default=0.1)) – Boosting learning rate. You can use callbacks parameter of fit method to shrink/adapt learning rate in training using reset_parameter callback. Note, that this will ignore the learning_rate argument in training.

- **n_estimators** (int, optional (default=100)) – Number of boosted trees to fit.

- **subsample_for_bin** (int, optional (default=200000)) – 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: ‘regression’ for LGBMRegressor, ‘binary’ or ‘multiclass’ for LGBMClassifier, ‘lambdarank’ for LGBMRanker.

- **class_weight** (dict, 'balanced' or None, optional (default=None)) – Weights associated with classes in the form {class_label: weight}. Use this parameter only for multi-class classification task; for binary classification task you may use is_unbalance or scale_pos_weight parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (https://scikit-learn.org/stable/modules/calibration.html) of your model. The ‘balanced’ mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes *
np.bincount(y)). If None, all classes are supposed to have weight one. Note, that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

- **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** (float, optional (default=1e-3)) – Minimum sum of instance weight (hessian) needed in a child (leaf).
- **min_child_samples** (int, optional (default=20)) – Minimum number of data needed in a child (leaf).
- **subsample** (float, optional (default=1.)) – Subsample ratio of the training instance.
- **subsample_freq** (int, optional (default=0)) – Frequency 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, RandomState object or None, optional (default=None)) – Random number seed. If int, this number is used to seed the C++ code. If RandomState object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.
- **n_jobs** (int, optional (default=-1)) – Number of parallel threads.
- **silent** (bool, optional (default=True)) – Whether to print messages while running boosting.
- **importance_type** (string, optional (default='split')) – The type of feature importance to be filled into feature_importances_. 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.
- **client** (dask.distributed.Client or None, optional (default=None)) – Dask client. If None, distributed.default_client() will be used at runtime. The Dask client used by this class will not be saved if the model object is pickled.
- ****kwargs – Other parameters for the model. Check http://lightgbm.readthedocs.io/en/latest/Parameters.html for more parameters.

**Warning:** **kwargs is not supported in sklearn, it may cause unexpected issues.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>Construct a gradient boosting model.</td>
</tr>
<tr>
<td><code>fit</code></td>
<td>Build a gradient boosting model from the training set (X, y).</td>
</tr>
<tr>
<td><code>get_params</code></td>
<td>Get parameters for this estimator.</td>
</tr>
<tr>
<td><code>predict</code></td>
<td>Return the predicted value for each sample.</td>
</tr>
<tr>
<td><code>set_params</code></td>
<td>Set the parameters of this estimator.</td>
</tr>
<tr>
<td><code>to_local</code></td>
<td>Create regular version of lightgbm.LGBMRanker from the distributed version.</td>
</tr>
</tbody>
</table>

## Attributes

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>best_iteration_</code></td>
<td>The best iteration of fitted model if <code>early_stopping_rounds</code> has been specified.</td>
</tr>
<tr>
<td><code>best_score_</code></td>
<td>The best score of fitted model.</td>
</tr>
<tr>
<td><code>booster_</code></td>
<td>The underlying Booster of this model.</td>
</tr>
<tr>
<td><code>client_</code></td>
<td>Dask client.</td>
</tr>
<tr>
<td><code>evals_result_</code></td>
<td>The evaluation results if <code>early_stopping_rounds</code> has been specified.</td>
</tr>
<tr>
<td><code>feature_importances_</code></td>
<td>The feature importances (the higher, the more important).</td>
</tr>
<tr>
<td><code>feature_name_</code></td>
<td>The names of features.</td>
</tr>
<tr>
<td><code>n_features_</code></td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td><code>n_features_in_</code></td>
<td>The number of features of fitted model.</td>
</tr>
<tr>
<td><code>objective_</code></td>
<td>The concrete objective used while fitting this model.</td>
</tr>
</tbody>
</table>

**property best_iteration_**

The best iteration of fitted model if `early_stopping_rounds` has been specified.

**Type** int or None

**property best_score_**

The best score of fitted model.

**Type** dict or None

**property booster_**

The underlying Booster of this model.

**Type** Booster

**property client_**

Dask client.

This property can be passed in the constructor or updated with `model.set_params(client=client)`. 

**Type** dask.distributed.Client

**property evals_result_**

The evaluation results if `early_stopping_rounds` has been specified.
**Type**  
dict or None

**property feature_importances_**  
The feature importances (the higher, the more important).

**Note:** importance_type attribute is passed to the function to configure the type of importance values to be extracted.

**Type**  
array of shape = [n_features]

**property feature_name_**  
The names of features.

**Type**  
array of shape = [n_features]

**fit**(X, y, sample_weight=None, init_score=None, group=None, **kwargs)  
Build a gradient boosting model from the training set (X, y).

**Parameters**

- **X** (Dask Array or Dask DataFrame of shape = [n_samples, n_features]) – Input feature matrix.
- **y** (Dask Array, Dask DataFrame or Dask Series of shape = [n_samples]) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (Dask Array, Dask DataFrame, Dask Series of shape = [n_samples] or None, optional (default=None)) – Weights of training data.
- **init_score** (Dask Array, Dask DataFrame, Dask Series of shape = [n_samples] or None, optional (default=None)) – Init score of training data.
- **group** (Dask Array, Dask DataFrame, Dask Series of shape = [n_samples] or None, optional (default=None)) – Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **verbose** (bool or int, optional (default=True)) – Requires at least one evaluation data. If True, the eval metric on the eval set is printed at each boosting stage. If int, the eval metric on the eval set is printed at every verbose boosting stage. The last boosting stage or the boosting stage found by using early_stopping_rounds is also printed.
Example

With \texttt{verbose} = 4 and at least one item in \texttt{eval_set}, an evaluation metric is printed every 4 (instead of 1) boosting stages.

- \texttt{feature_name} \hspace{1cm} (list of strings or 'auto', optional (default='auto')) – Feature names. If ‘auto’ and data is pandas DataFrame, data columns names are used.

- \texttt{categorical_feature} \hspace{1cm} (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 \texttt{feature_name} as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature.

- \texttt{**kwargs} – Other parameters passed through to \texttt{LGBMRanker.fit()}.

\texttt{get_params (deep=True)}

Get parameters for this estimator.

Parameters \texttt{deep} (bool, optional (default=True)) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns \texttt{params} – Parameter names mapped to their values.

Return type dict

\texttt{property n_features_}

The number of features of fitted model.

Type int

\texttt{property n_features_in_}

The number of features of fitted model.

Type int

\texttt{property objective_}

The concrete objective used while fitting this model.

Type string or callable

\texttt{predict (X, **kwargs)}

Return the predicted value for each sample.

Parameters

- \texttt{X} \hspace{1cm} (Dask Array or Dask DataFrame of shape = [n_samples, n_features]) – Input features matrix.

- \texttt{raw_score} (bool, optional (default=False)) – Whether to predict raw scores.

- \texttt{start_iteration} (int, optional (default=0)) – Start index of the iteration to predict. If <= 0, starts from the first iteration.

- \texttt{num_iteration} (int or None, optional (default=None)) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration <= 0, the best iteration is used; otherwise, all iterations
from `start_iteration` are used (no limits). If <= 0, all iterations from `start_iteration` are used (no limits).

- **pred_leaf** *(bool, optional (default=False))* – Whether to predict leaf index.

- **pred_contrib** *(bool, optional (default=False))* – Whether to predict feature contributions.

**Note:** If you want to get more explanations for your model’s predictions using SHAP values, like SHAP interaction values, you can install the shap package (https://github.com/slundberg/shap). Note that unlike the shap package, with `pred_contrib` we return a matrix with an extra column, where the last column is the expected value.

- ****kwargs – Other parameters for the prediction.

**Returns**

- **predicted_result** *(Dask Array of shape = [n_samples]*) – The predicted values.

- **X_leaves** *(Dask Array of shape = [n_samples, n_trees]*) – If `pred_leaf=True`, the predicted leaf of every tree for each sample.

- **X_SHAP_values** *(Dask Array of shape = [n_samples, n_features + 1]*) – If `pred_contrib=True`, the feature contributions for each sample.

**set_params**( **params**)

Set the parameters of this estimator.

**Parameters** **params** – Parameter names with their new values.

**Returns** **self** – Returns self.

**Return type** object

**to_local**( )

Create regular version of lightgbm.LGBMRanker from the distributed version.

**Returns** **model** – Local underlying model.

**Return type** `lightgbm.LGBMRanker`

### 9.5 Callbacks

<table>
<thead>
<tr>
<th><strong>Callback</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>early_stopping</code> <em>(stopping_rounds[, ...]</em>)</td>
<td>Create a callback that activates early stopping.</td>
</tr>
<tr>
<td><code>print_evaluation</code> <em>(period, show_stdv)</em></td>
<td>Create a callback that prints the evaluation results.</td>
</tr>
<tr>
<td><code>record_evaluation</code> <em>(eval_result)</em></td>
<td>Create a callback that records the evaluation history into <code>eval_result</code>.</td>
</tr>
<tr>
<td><code>reset_parameter</code> *(*<em>kwargs)</em></td>
<td>Create a callback that resets the parameter after the first iteration.</td>
</tr>
</tbody>
</table>
9.5.1 lightgbm.early_stopping

`lightgbm.early_stopping(stopping_rounds, first_metric_only=False, verbose=True)`
Create a callback that activates early stopping.

Activates early stopping. The model will train until the validation score stops improving. Validation score needs to improve at least every `early_stopping_rounds` round(s) to continue training. Requires at least one validation data and one metric. If there’s more than one, will check all of them. But the training data is ignored anyway. To check only the first metric set `first_metric_only` to True.

**Parameters**

- **stopping_rounds (int)** – The possible number of rounds without the trend occurrence.
- **first_metric_only (bool, optional (default=False))** – Whether to use only the first metric for early stopping.
- **verbose (bool, optional (default=True))** – Whether to print message with early stopping information.

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

**Return type** function

9.5.2 lightgbm.print_evaluation

`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

9.5.3 lightgbm.record_evaluation

`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
9.5.4 lightgbm.reset_parameter

```
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

9.6 Plotting

```
plot_importance(booster[, ax, height, xlim, ...]) Plot model’s feature importances.
plot_split_value_histogram(booster, feature) Plot split value histogram for the specified feature of the model.
plot_metric(booster[, metric, ...]) Plot one metric during training.
plot_tree(booster[, ax, tree_index, ...]) Plot specified tree.
create_tree_digraph(booster[, tree_index, ...]) Create a digraph representation of specified tree.
```

9.6.1 lightgbm.plot_importance

```
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, dpi=None, grid=True, precision=3, **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.

• dpi (int or None, optional (default=None)) – Resolution of the figure.

• grid (bool, optional (default=True)) – Whether to add a grid for axes.

• precision (int or None, optional (default=3)) – Used to restrict the display of floating point values to a certain precision.

• **kwargs – Other parameters passed to ax.barh().

Returns ax – The plot with model’s feature importances.

Return type matplotlib.axes.Axes

### 9.6.2 `lightgbm.plot_split_value_histogram`

`lightgbm.plot_split_value_histogram` (booster, feature, bins=None, ax=None, width_coef=0.8, xlim=None, ylim=None, title='split value histogram for feature with @index/name@ @feature@', xlabel='Feature split value', ylabel='Count', figsize=None, dpi=None, grid=True, **kwargs)

Plot split value histogram for the specified feature of the model.

Parameters

• booster (Booster or LGBMModel) – Booster or LGBMModel instance of which feature split value histogram should be plotted.

• feature (int or string) – The feature name or index the histogram is plotted for. If int, interpreted as index. If string, interpreted as name.

• bins (int, string or None, optional (default=None)) – The maximum number of bins. If None, the number of bins equals number of unique split values. If string, it should be one from the list of the supported values by `numpy.histogram()` function.

• ax (matplotlib.axes.Axes or None, optional (default=None)) – Target axes instance. If None, new figure and axes will be created.

• width_coef (float, optional (default=0.8)) – Coefficient for histogram bar width.

• 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="Split value histogram for feature with @index/name@ @feature@")) – Axes title. If None, title is disabled. @feature@ placeholder can be used, and it will be replaced with the value of feature parameter. @index/name@ placeholder can be used, and it will be replaced with index word in case of int type feature parameter or name word in case of string type feature parameter.

**xlabel** (string or None, optional (default="Feature split value")) – X-axis title label. If None, title is disabled.

**ylabel** (string or None, optional (default="Count")) – Y-axis title label. If None, title is disabled.

**figsize** (tuple of 2 elements or None, optional (default=None)) – Figure size.

**dpi** (int or None, optional (default=None)) – Resolution of the figure.

**grid** (bool, optional (default=True)) – Whether to add a grid for axes.

****kwargs** – Other parameters passed to ax.bar().

Returns ax – The plot with specified model’s feature split value histogram.

Return type matplotlib.axes.Axes

### 9.6.3 lightgbm.plot_metric

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, dpi=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.
LightGBM, Release 3.2.0.99

- `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.*
- `dpi` *(int or None, optional (default=None)) – Resolution of the figure.*
- `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`

9.6.4 `lightgbm.plot_tree`

`lightgbm.plot_tree` *(booster, ax=None, tree_index=0, figsize=None, dpi=None, show_info=None, precision=3, orientation='horizontal', **kwargs)*

Plot specified tree.

Each node in the graph represents a node in the tree.

Non-leaf nodes have labels like `Column_10 <= 875.9`, which means “this node splits on the feature named “Column_10”, with threshold 875.9”.

Leaf nodes have labels like `leaf 2: 0.422`, which means “this node is a leaf node, and the predicted value for records that fall into this node is 0.422”. The number (2) is an internal unique identifier and doesn’t have any special meaning.

**Note:** It is preferable to use `create_tree_digraph()` because of its lossless quality and returned objects can be also rendered and displayed directly inside a Jupyter notebook.

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.*
- `dpi` *(int or None, optional (default=None)) – Resolution of the figure.*
- `show_info` *(list of strings or None, optional (default=None)) – What information should be shown in nodes.
  - 'split_gain': gain from adding this split to the model
  - 'internal_value': raw predicted value that would be produced by this node if it was a leaf node
  - 'internal_count': number of records from the training data that fall into this non-leaf node
- 'internal_weight': total weight of all nodes that fall into this non-leaf node
- 'leaf_count': number of records from the training data that fall into this leaf node
- 'internal_weight': total weight (sum of hessian) of all observations that fall into this leaf node
- 'data_percentage': percentage of training data that fall into this node


Returns ax – The plot with single tree.

Return type matplotlib.axes.Axes

9.6.5 lightgbm.create_tree_digraph

lightgbm.create_tree_digraph(booster, tree_index=0, show_info=None, precision=3, orientation='horizontal', **kwargs)

Create a digraph representation of specified tree.

Each node in the graph represents a node in the tree.

Non-leaf nodes have labels like Column_10 <= 875.9, which means “this node splits on the feature named “Column_10”, with threshold 875.9”.

Leaf nodes have labels like leaf 2: 0.422, which means “this node is a leaf node, and the predicted value for records that fall into this node is 0.422”. The number (2) is an internal unique identifier and doesn’t have any special meaning.


Parameters

- booster(Booster or LGBMModel) – Booster or LGBMModel instance to be converted.
- tree_index (int, optional (default=0)) – The index of a target tree to convert.
- show_info (list of strings or None, optional (default=None)) – What information should be shown in nodes.
  - 'split_gain': gain from adding this split to the model
  - 'internal_value': raw predicted value that would be produced by this node if it was a leaf node
  - 'internal_count': number of records from the training data that fall into this non-leaf node
  - 'internal_weight': total weight of all nodes that fall into this non-leaf node

9.6. Plotting
- 'leaf_count': number of records from the training data that fall into this leaf node
- 'leaf_weight': total weight (sum of hessian) of all observations that fall into this leaf node
- 'data_percentage': percentage of training data that fall into this node

**precision** *(int or None, optional (default=3)) – Used to restrict the display of floating point values to a certain precision.*

**orientation** *(string, optional (default='horizontal')) – Orientation of the tree. Can be ‘horizontal’ or ‘vertical’.*

**kwargs** – Other parameters passed to Digraph constructor. Check https://graphviz.readthedocs.io/en/stable/api.html#digraph for the full list of supported parameters.

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

Return type: **graphviz.Digraph**

### 9.7 Utilities

| register_logger(logger) | Register custom logger. |

#### 9.7.1 lightgbm.register_logger

lightgbm.**register_logger**(logger)

Register custom logger.

**Parameters**

- **logger** *(logging.Logger)* – Custom logger.
CHAPTER
TEN

DISTRIBUTED LEARNING GUIDE

This guide describes distributed learning in LightGBM. Distributed learning allows the use of multiple machines to produce a single model.

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

10.1 How Distributed LightGBM Works

This section describes how distributed learning in LightGBM works. To learn how to do this in various programming languages and frameworks, please see Integrations.

10.1.1 Choose Appropriate Parallel Algorithm

LightGBM provides 3 distributed learning algorithms now.

<table>
<thead>
<tr>
<th>Parallel Algorithm</th>
<th>How to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data parallel</td>
<td>tree_learner=data</td>
</tr>
<tr>
<td>Feature parallel</td>
<td>tree_learner=feature</td>
</tr>
<tr>
<td>Voting parallel</td>
<td>tree_learner=voting</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>#feature is small</th>
<th>#data is small</th>
<th>#feature is large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Parallel</td>
<td>Data Parallel</td>
<td>Voting Parallel</td>
</tr>
</tbody>
</table>

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

10.2 Integrations

This section describes how to run distributed LightGBM training in various programming languages and frameworks. To learn how distributed learning in LightGBM works generally, please see How Distributed LightGBM Works.
10.2.1 Apache Spark

Apache Spark users can use MMLSpark for machine learning workflows with LightGBM. This project is not maintained by LightGBM’s maintainers.

See this MMLSpark example and the MMLSpark documentation for additional information on using LightGBM on Spark.

Note: MMLSpark is not maintained by LightGBM’s maintainers. Bug reports or feature requests should be directed to https://github.com/Azure/mmlspark/issues.

10.2.2 Dask

New in version 3.2.0.

LightGBM’s Python package supports distributed learning via Dask. This integration is maintained by LightGBM’s maintainers.

Warning: Dask integration is only tested on Linux.

Dask Examples

For sample code using lightgbm.dask, see these Dask examples.

Training with Dask

This section contains detailed information on performing LightGBM distributed training using Dask.

Configuring the Dask Cluster

Allocating Threads

When setting up a Dask cluster for training, give each Dask worker process at least two threads. If you do not do this, training might be substantially slower because communication work and training work will block each other.

If you do not have other significant processes competing with Dask for resources, just accept the default $n\text{threads}$ from your chosen dask.distributed.cluster.

```python
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=3)
client = Client(cluster)
```

Managing Memory

Use the Dask diagnostic dashboard or your preferred monitoring tool to monitor Dask workers’ memory consumption during training. As described in the Dask worker documentation, Dask workers will automatically start spilling data to disk if memory consumption gets too high. This can substantially slow down computations, since disk I/O is usually much slower than reading the same data from memory.

At 60% of memory load, [Dask will] spill least recently used data to disk
To reduce the risk of hitting memory limits, consider restarting each worker process before running any data loading or training code.

```python
client.restart()
```

### Setting Up Training Data

The estimators in `lightgbm.dask` expect that matrix-like or array-like data are provided in Dask DataFrame, Dask Array, or (in some cases) Dask Series format. See the Dask DataFrame documentation and the Dask Array documentation for more information on how to create such data structures.

While setting up for training, `lightgbm` will concatenate all of the partitions on a worker into a single dataset. Distributed training then proceeds with one LightGBM worker process per Dask worker.

When setting up data partitioning for LightGBM training with Dask, try to follow these suggestions:

- ensure that each worker in the cluster has some of the training data
- try to give each worker roughly the same amount of data, especially if your dataset is small
- if you plan to train multiple models (for example, to tune hyperparameters) on the same data, use `client.persist()` before training to materialize the data one time

### Using a Specific Dask Client

In most situations, you should not need to tell `lightgbm.dask` to use a specific Dask client. By default, the client returned by `distributed.default_client()` will be used.

However, you might want to explicitly control the Dask client used by LightGBM if you have multiple active clients in the same session. This is useful in more complex workflows like running multiple training jobs on different Dask clusters.

LightGBM’s Dask estimators support setting an attribute `client` to control the client that is used.

```python
import lightgbm as lgb
from distributed import Client, LocalCluster

cluster = LocalCluster()
client = Client(cluster)

# option 1: keyword argument in constructor
dask_model = lgb.DaskLGBMClassifier(client=client)

# option 2: set_params() after construction
dask_model = lgb.DaskLGBMClassifier()
dask_model.set_params(client=client)
```
Using Specific Ports

At the beginning of training, `lightgbm.dask` sets up a LightGBM network where each Dask worker runs one long-running task that acts as a LightGBM worker. During training, LightGBM workers communicate with each other over TCP sockets. By default, random open ports are used when creating these sockets.

If the communication between Dask workers in the cluster used for training is restricted by firewall rules, you must tell LightGBM exactly what ports to use.

Option 1: provide a specific list of addresses and ports

LightGBM supports a parameter `machines`, a comma-delimited string where each entry refers to one worker (host name or IP) and a port that that worker will accept connections on. If you provide this parameter to the estimators in `lightgbm.dask`, LightGBM will not search randomly for ports.

For example, consider the case where you are running one Dask worker process on each of the following IP addresses:

```
10.0.1.0
10.0.2.0
10.0.3.0
```

You could edit your firewall rules to allow traffic on one additional port on each of these hosts, then provide `machines` directly.

```python
import lightgbm as lgb
machines = "10.0.1.0:12401,10.0.2.0:12402,10.0.3.0:15000"
dask_model = lgb.DaskLGBMRegressor(machines=machines)
```

If you are running multiple Dask worker processes on physical host in the cluster, be sure that there are multiple entries for that IP address, with different ports. For example, if you were running a cluster with `nprocs=2` (2 Dask worker processes per machine), you might open two additional ports on each of these hosts, then provide `machines` as follows.

```python
import lightgbm as lgb
machines = ",".join([
    "10.0.1.0:16000",
    "10.0.1.0:16001",
    "10.0.2.0:16000",
    "10.0.2.0:16001",
])
dask_model = lgb.DaskLGBMRegressor(machines=machines)
```

**Warning:** Providing `machines` gives you complete control over the networking details of training, but it also makes the training process fragile. Training will fail if you use `machines` and any of the following are true:

- any of the ports mentioned in `machines` are not open when training begins
- some partitions of the training data are held by machines that that are not present in `machines`
- some machines mentioned in `machines` do not hold any of the training data

Option 2: specify one port to use on every worker

If you are only running one Dask worker process on each host, and if you can reliably identify a port that is open on every host, using `machines` is unnecessarily complicated. If `local_listen_port` is given and `machines` is
not, LightGBM will not search for ports randomly, but it will limit the list of addresses in the LightGBM network to those Dask workers that have a piece of the training data.

For example, consider the case where you are running one Dask worker process on each of the following IP addresses:

<table>
<thead>
<tr>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.1.0</td>
</tr>
<tr>
<td>10.0.2.0</td>
</tr>
<tr>
<td>10.0.3.0</td>
</tr>
</tbody>
</table>

You could edit your firewall rules to allow communication between any of the workers over one port, then provide that port via parameter `local_listen_port`.

```python
import lightgbm as lgb

dask_model = lgb.DaskLGBMRegressor(local_listen_port=12400)
```

**Warning:** Providing `local_listen_port` is slightly less fragile than `machines` because LightGBM will automatically figure out which workers have pieces of the training data. However, using this method, training can fail if any of the following are true:

- the port `local_listen_port` is not open on any of the worker hosts
- any machine has multiple Dask worker processes running on it

### Prediction with Dask

The estimators from `lightgbm.dask` can be used to create predictions based on data stored in Dask collections. In that interface, `.predict()` expects a Dask Array or Dask DataFrame, and returns a Dask Array of predictions.

See the [Dask prediction example](#) for some sample code that shows how to perform Dask-based prediction.

For model evaluation, consider using the metrics functions from `dask-ml`. Those functions are intended to provide the same API as equivalent functions in `sklearn.metrics`, but they use distributed computation powered by Dask to compute metrics without all of the input data ever needing to be on a single machine.

### Saving Dask Models

After training with Dask, you have several options for saving a fitted model.

**Option 1: pickle the Dask estimator**

LightGBM's Dask estimators can be pickled directly with `cloudpickle`, `joblib`, or `pickle`.

```python
import dask.array as da
import pickle
import lightgbm as lgb
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=2)
client = Client(cluster)

X = da.random.random((1000, 10), (500, 10))
y = da.random.random((1000,), (500,))

dask_model = lgb.DaskLGBMRegressor()
```
dask_model.fit(X, y)

with open("dask-model.pkl", "wb") as f:
    pickle.dump(dask_model, f)

A model saved this way can then later be loaded with whichever serialization library you used to save it.

```python
import pickle
with open("dask-model.pkl", "rb") as f:
    dask_model = pickle.load(f)
```

**Note:** If you explicitly set a Dask client (see Using a Specific Dask Client), it will not be saved when pickling the estimator. When loading a Dask estimator from disk, if you need to use a specific client you can add it after loading with `dask_model.set_params(client=client)`.

**Option 2: pickle the sklearn estimator**

The estimators available from `lightgbm.dask` can be converted to an instance of the equivalent class from `lightgbm.sklearn`. Choosing this option allows you to use Dask for training but avoid depending on any Dask libraries at scoring time.

```python
import dask.array as da
import joblib
import lightgbm as lgb
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=2)
client = Client(cluster)

X = da.random.random((1000, 10), (500, 10))
y = da.random.random((1000), (500,))

dask_model = lgb.DaskLGBMRegressor()
dask_model.fit(X, y)

# convert to sklearn equivalent
sklearn_model = dask_model.to_local()

print(type(sklearn_model))
#> lightgbm.sklearn.LGBMRegressor

joblib.dump(sklearn_model, "sklearn-model.joblib")

A model saved this way can then later be loaded with whichever serialization library you used to save it.

```python
import joblib

sklearn_model = joblib.load("sklearn-model.joblib")
```

**Option 3: save the LightGBM Booster**

The lowest-level model object in LightGBM is the `lightgbm.Booster`. After training, you can extract a Booster from the Dask estimator.
```python
import dask.array as da
import lightgbm as lgb
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=2)
client = Client(cluster)

X = da.random.random((1000, 10), (500, 10))
y = da.random.random((1000,), (500,))

dask_model = lgb.DaskLGBMRegressor()
dask_model.fit(X, y)

# get underlying Booster object
bst = dask_model.booster_
```

From the point forward, you can use any of the following methods to save the Booster:

- **serialize** with `cloudpickle`, `joblib`, or `pickle`  
- `bst.dump_model()`: dump the model to a dictionary which could be written out as JSON  
- `bst.model_to_string()`: dump the model to a string in memory  
- `bst.save_model()`: write the output of `bst.model_to_string()` to a text file

### 10.2.3 Kubeflow

Kubeflow Fairing supports LightGBM distributed training. These examples show how to get started with LightGBM and Kubeflow Fairing in a hybrid cloud environment.

Kubeflow users can also use the Kubeflow XGBoost Operator for machine learning workflows with LightGBM. You can see this example for more details.

Kubeflow integrations for LightGBM are not maintained by LightGBM’s maintainers.

**Note:** The Kubeflow integrations for LightGBM are not maintained by LightGBM’s maintainers. Bug reports or feature requests should be directed to https://github.com/kubeflow/fairing/issues or https://github.com/kubeflow/xgboost-operator/issues.

### 10.2.4 LightGBM CLI

**Preparation**

By default, distributed learning with LightGBM uses socket-based communication.

If you need to build distributed version with MPI support, please refer to Installation Guide.
Socket Version

It needs to collect IP of all machines that want to run distributed 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 distributed 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 Distributed 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 distributed learning example
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.

11.1 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 an AMD GPU, you should download and install the AMDGPU-Pro driver and also install package ocl-icd-libopencl1 and ocl-icd-opencl-dev.

11.2 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):
Now we are ready to checkout LightGBM and compile it with GPU support:

```bash
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:

```

```

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

```bash
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
mkdir build
cd build
```

```bash
# cmake -DUSE_GPU=1 -DOpenCL_LIBRARY=/usr/local/cuda/lib64/libOpenCL.so -DOpenCL_INCLUDE_DIR=/usr/local/cuda/include/..
make -j$(nproc)
```

```bash
```

You will see two binaries are generated, `lightgbm` and `lib_lightgbm.so`. If you are building on macOS, 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.

### 11.3 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):

```bash
sudo apt-get -y install python-pip
```n

```
```

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-package Examples for more information on how to use the Python interface.

### 11.4 Dataset Preparation

Using the following commands to prepare the Higgs dataset:

```bash
git clone https://github.com/guolinke/boosting_tree_benchmarks.git
cd boosting_tree_benchmarks/data
gunzip HIGGS.csv.gz
python higgs2libsvm.py
```

```bash
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):

```bash
cd /mnt/workspace
```

```bash
```

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

```bash
```

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

```bash
```

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

```bash
```

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

```bash
```
GPU is enabled in the configuration file we just created by setting device=gpu. In this configuration we use the first GPU installed on the system (gpu_platform_id=0 and gpu_device_id=0). If gpu_platform_id or gpu_device_id is not set, the default platform and GPU will be selected. You might have multiple platforms (AMD/Intel/NVIDIA) or GPUs. You can use the clinfo utility to identify the GPUs on each platform. On Ubuntu, you can install clinfo by executing:

```
sudo apt-get install clinfo
```

If you have a discrete GPU by AMD/NVIDIA and an integrated GPU by Intel, make sure to select the correct gpu_platform_id to use the discrete GPU.

### 11.5 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
```

---

**11.5. Run Your First Learning Task on GPU** 181
Also, you can compare the training speed with CPU:

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

### 11.6 Further Reading

- GPU Tuning Guide and Performance Comparison
- GPU SDK Correspondence and Device Targeting Table
- GPU Windows Tutorial

### 11.7 Reference

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

12.1 Missing Value Handle

- LightGBM enables the missing value handle by default. Disable it by setting `use_missing=false`.
- LightGBM uses NA (NaN) to represent missing values by default. Change it to use zero by setting `zero_as_missing=true`.
- When `zero_as_missing=false` (default), the unshown values in sparse matrices (and LightSVM) are treated as zeros.
- When `zero_as_missing=true`, NA and zeros (including unshown values in sparse matrices (and LightSVM)) are treated as missing.

12.2 Categorical Feature Support

- LightGBM offers good accuracy with integer-encoded categorical features. LightGBM applies Fisher (1958) to find the optimal split over categories as described here. This often performs better than one-hot encoding.
- Use `categorical_feature` to specify the categorical features. Refer to the parameter `categorical_feature` in Parameters.
- Categorical features must be encoded as non-negative integers (int) less than `Int32.MaxValue` (2147483647). It is best to use a contiguous range of integers started from zero.
- Use `min_data_per_group, cat_smooth` to deal with over-fitting (when #data is small or #category is large).
- For a categorical feature with high cardinality (#category is large), it often works best to treat the feature as numeric, either by simply ignoring the categorical interpretation of the integers or by embedding the categories in a low-dimensional numeric space.

12.3 LambdaRank

- The label should be of type int, such that larger numbers correspond to higher relevance (e.g. 0:bad, 1:fair, 2:good, 3:perfect).
- Use `label_gain` to set the gain(weight) of int label.
- Use `lambdarank_truncation_level` to truncate the max DCG.
12.4 Cost Efficient Gradient Boosting

Cost Efficient Gradient Boosting (CEGB) makes it possible to penalise boosting based on the cost of obtaining feature values. CEGB penalises learning in the following ways:

- Each time a tree is split, a penalty of `cegb_penalty_split` is applied.
- When a feature is used for the first time, `cegb_penalty_feature_coupled` is applied. This penalty can be different for each feature and should be specified as one double per feature.
- When a feature is used for the first time for a data row, `cegb_penalty_feature_lazy` is applied. Like `cegb_penalty_feature_coupled`, this penalty is specified as one double per feature.

Each of the penalties above is scaled by `cegb_tradeoff`. Using this parameter, it is possible to change the overall strength of the CEGB penalties by changing only one parameter.

12.5 Parameters Tuning

- Refer to Parameters Tuning.

12.6 Distributed Learning

- Refer to Distributed Learning Guide.

12.7 GPU Support

- Refer to GPU Tutorial and GPU Targets.

12.8 Recommendations for gcc Users (MinGW, *nix)

- Refer to gcc Tips.
CHAPTER
THIRTEEN

LIGHTGBM FAQ

13.1 Critical Issues

A critical issue could be a crash, prediction error, nonsense output, or something else requiring immediate attention. Please post such an issue in the Microsoft/LightGBM repository.

You may also ping a member of the core team according to the relevant area of expertise by mentioning them with the arabase (@) symbol:

- @guolinke Guolin Ke (C++ code / R-package / Python-package)
- @chivee Qiwei Ye (C++ code / Python-package)
- @shiyu1994 Yu Shi (C++ code / Python-package)
- @brotta Belinda Trotta (C++ code)
- @Laurae2 Damien Soukhavong (R-package)
- @jameslamb James Lamb (R-package / Dask-package)
- @wxchan Wenxuan Chen (Python-package)
- @henry0312 Tsukasa Omoto (Python-package)
- @StrikerRUS Nikita Titov (Python-package)
- @huanzhang12 Huan Zhang (GPU support)

Please include as much of the following information as possible when submitting a critical issue:

- Is it reproducible on CLI (command line interface), R, and/or Python?
- Is it specific to a wrapper? (R or Python?)
13.2 General LightGBM Questions

• 1. Where do I find more details about LightGBM parameters?
• 2. On datasets with millions of features, training does not start (or starts after a very long time).
• 3. When running LightGBM on a large dataset, my computer runs out of RAM.
• 4. I am using Windows. Should I use Visual Studio or MinGW for compiling LightGBM?
• 5. When using LightGBM GPU, I cannot reproduce results over several runs.
• 6. Bagging is not reproducible when changing the number of threads.
• 7. I tried to use Random Forest mode, and LightGBM crashes!
• 8. CPU usage is low (like 10%) in Windows when using LightGBM on very large datasets with many-core systems.
• 9. When I’m trying to specify a categorical column with the `categorical_feature` parameter, I get the following sequence of warnings, but there are no negative values in the column.
• 10. LightGBM crashes randomly with the error like: Initializing libomp5.dylib, but found libomp.dylib already initialized.
• 11. LightGBM hangs when multithreading (OpenMP) and using forking in Linux at the same time.
• 12. Why is early stopping not enabled by default in LightGBM?
• 13. Does LightGBM support direct loading data from zero-based or one-based LibSVM format file?
• 14. Why CMake cannot find the compiler when compiling LightGBM with MinGW?
• 15. Where can I find LightGBM’s logo to use it in my presentation?

13.2.1 1. Where do I find more details about LightGBM parameters?

Take a look at Parameters and the Laurae++/Parameters website.
13.2.2 2. On datasets with millions of features, training does not start (or starts after a very long time).

Use a smaller value for \texttt{bin\_construct\_sample\_cnt} and a larger value for \texttt{min\_data}.

13.2.3 3. When running LightGBM on a large dataset, my computer runs out of RAM.

\textbf{Multiple Solutions}: set the \texttt{histogram\_pool\_size} parameter to the MB you want to use for LightGBM (\texttt{histogram\_pool\_size} + dataset size = approximately RAM used), lower \texttt{num\_leaves} or lower \texttt{max\_bin} (see Microsoft/LightGBM#562).

13.2.4 4. I am using Windows. Should I use Visual Studio or MinGW for compiling LightGBM?

Visual Studio performs best for LightGBM.

13.2.5 5. When using LightGBM GPU, I cannot reproduce results over several runs.

This is normal and expected behaviour, but you may try to use \texttt{gpu\_use\_dp = true} for reproducibility (see Microsoft/LightGBM#560). You may also use the CPU version.

13.2.6 6. Bagging is not reproducible when changing the number of threads.

LightGBM bagging is multithreaded, so its output depends on the number of threads used. There is no workaround currently.

Starting from #2804 bagging result doesn’t depend on the number of threads. So this issue should be solved in the latest version.

13.2.7 7. I tried to use Random Forest mode, and LightGBM crashes!

This is expected behaviour for arbitrary parameters. To enable Random Forest, you must use \texttt{bagging\_fraction} and \texttt{feature\_fraction} different from 1, along with a \texttt{bagging\_freq}. This thread includes an example.

13.2.8 8. CPU usage is low (like 10%) in Windows when using LightGBM on very large datasets with many-core systems.

Please use Visual Studio as it may be 10x faster than MinGW especially for very large trees.
13.2.9 9. When I’m trying to specify a categorical column with the `categorical_feature` parameter, I get the following sequence of warnings, but there are no negative values in the column.

```
[LightGBM] [Warning] Met negative value in categorical features, will convert it to NaN.
[LightGBM] [Warning] There are no meaningful features, as all feature values are constant.
```

The column you’re trying to pass via `categorical_feature` likely contains very large values. Categorical features in LightGBM are limited by int32 range, so you cannot pass values that are greater than Int32.MaxValue (2147483647) as categorical features (see Microsoft/LightGBM#1359). You should convert them to integers ranging from zero to the number of categories first.

13.2.10 10. LightGBM crashes randomly with the error like:

```
Initializing libiomp5.dylib, but found libomp.dylib already initialized.
```

Possible Cause: This error means that you have multiple OpenMP libraries installed on your machine and they conflict with each other. (File extensions in the error message may differ depending on the operating system).

If you are using Python distributed by Conda, then it is highly likely that the error is caused by the `numpy` package from Conda which includes the `mkl` package which in turn conflicts with the system-wide library. In this case you can update the `numpy` package in Conda or replace the Conda’s OpenMP library instance with system-wide one by creating a symlink to it in Conda environment folder `$CONDA_PREFIX/lib`.

Solution: Assuming you are using macOS with Homebrew, the command which overwrites OpenMP library files in the current active Conda environment with symlinks to the system-wide library ones installed by Homebrew:

```
ln -sf `ls -d "$\$(brew --cellar libomp)/*/lib`/* $CONDA_PREFIX/lib
```

The described above fix worked fine before the release of OpenMP 8.0.0 version. Starting from 8.0.0 version, Homebrew formula for OpenMP includes `-DLIBOMP_INSTALL_ALIASES=OFF` option which leads to that the fix doesn’t work anymore. However, you can create symlinks to library aliases manually:

```
for LIBOMP_ALIAS in libgomp.dylib libomp5.dylib libomp.dylib; do sudo ln -sf "$\$(brew --cellar libomp --cellar libomp)/*/lib/libomp.dylib $CONDA_PREFIX/lib/$LIBOMP_ALIAS; done
```

Another workaround would be removing MKL optimizations from Conda’s packages completely:

```
conda install nomkl
```
If this is not your case, then you should find conflicting OpenMP library installations on your own and leave only one of them.

13.2.11 11. LightGBM hangs when multithreading (OpenMP) and using forking in Linux at the same time.

Use \texttt{nthreads=1} to disable multithreading of LightGBM. There is a bug with OpenMP which hangs forked sessions with multithreading activated. A more expensive solution is to use new processes instead of using fork, however, keep in mind it is creating new processes where you have to copy memory and load libraries (example: if you want to fork 16 times your current process, then you will require to make 16 copies of your dataset in memory) (see Microsoft/LightGBM#1789).

An alternative, if multithreading is really necessary inside the forked sessions, would be to compile LightGBM with Intel toolchain. Intel compilers are unaffected by this bug.

For C/C++ users, any OpenMP feature cannot be used before the fork happens. If an OpenMP feature is used before the fork happens (example: using OpenMP for forking), OpenMP will hang inside the forked sessions. Use new processes instead and copy memory as required by creating new processes instead of forking (or, use Intel compilers).

Cloud platform container services may cause LightGBM to hang, if they use Linux fork to run multiple containers on a single instance. For example, LightGBM hangs in AWS Batch array jobs, which use the ECS agent to manage multiple running jobs. Setting \texttt{nthreads=1} mitigates the issue.

13.2.12 12. Why is early stopping not enabled by default in LightGBM?

Early stopping involves choosing a validation set, a special type of holdout which is used to evaluate the current state of the model after each iteration to see if training can stop.

In LightGBM, we have decided to require that users specify this set directly. Many options exist for splitting training data into training, test, and validation sets.

The appropriate splitting strategy depends on the task and domain of the data, information that a modeler has but which LightGBM as a general-purpose tool does not.

13.2.13 13. Does LightGBM support direct loading data from zero-based or one-based LibSVM format file?

LightGBM supports loading data from zero-based LibSVM format file directly.

13.2.14 14. Why CMake cannot find the compiler when compiling LightGBM with MinGW?

\begin{verbatim}
CMake Error: CMAKE_C_COMPILER not set, after EnableLanguage
CMake Error: CMAKE_CXX_COMPILER not set, after EnableLanguage
\end{verbatim}

This is a known issue of CMake when using MinGW. The easiest solution is to run again your \texttt{cmake} command to bypass the one time stopper from CMake. Or you can upgrade your version of CMake to at least version 3.17.0.

See Microsoft/LightGBM#3060 for more details.
13.2.15 15. Where can I find LightGBM’s logo to use it in my presentation?

You can find LightGBM’s logo in different file formats and resolutions here.

13.3 R-package

- 1. Any training command using LightGBM does not work after an error occurred during the training of a previous LightGBM model.
- 2. I used setinfo(), tried to print my lgb.Dataset, and now the R console froze!
- 3. error in data.table::data.table()...argument 2 is NULL

13.3.1 1. Any training command using LightGBM does not work after an error occurred during the training of a previous LightGBM model.

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.

13.3.2 2. I used setinfo(), tried to print my lgb.Dataset, and now the R console froze!

Avoid printing the lgb.Dataset after using setinfo. This is a known bug: Microsoft/LightGBM#539.

13.3.3 3. error in data.table::data.table()...argument 2 is NULL

If you are experiencing this error when running lightgbm, you may be facing the same issue reported in #2715 and later in #2989. We have seen that some in some situations, using data.table 1.11.x results in this error. To get around this, you can upgrade your version of data.table to at least version 1.12.0.

13.4 Python-package

- 1. Error: setup script specifies an absolute path when installing from GitHub using python setup.py install.
- 2. Error messages: Cannot ... before construct dataset.
- 3. I encounter segmentation faults (segfaults) randomly after installing LightGBM from PyPI using pip install lightgbm.
13.4.1 1. Error: setup script specifies an absolute path when installing 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.
```

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.

13.4.2 2. Error messages: Cannot ... before construct dataset.

I see error messages like...
```
Cannot get/set label/weight/init_score/group/num_data/num_feature before construct_dataset
```
but I’ve already constructed a 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: 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 constructing a Booster. 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/data) before constructing a dataset, it’s same as get self.label;
- set label (or weight/init_score/group) before constructing a dataset, it’s same as self.label=some_label_array;
- get num_data (or num_feature) before constructing a dataset, you can get data with self.data. Then, if your data is numpy.ndarray, use some code like self.data.shape. But do not do this after subsetting the Dataset, because you’ll get always None;
- set predictor (or reference/categorical feature) after constructing a dataset, you should set free_raw_data=False or init a Dataset object with the same raw data.

13.4.3 3. I encounter segmentation faults (segfaults) randomly after installing LightGBM from PyPI using pip install lightgbm.

We are doing our best to provide universal wheels which have high running speed and are compatible with any hardware, OS, compiler, etc. at the same time. However, sometimes it’s just impossible to guarantee the possibility of usage of LightGBM in any specific environment (see Microsoft/LightGBM#1743).

Therefore, the first thing you should try in case of segfaults is compiling from the source using pip install --no-binary :all: lightgbm. For the OS-specific prerequisites see this guide.

Also, feel free to post a new issue in our GitHub repository. We always look at each case individually and try to find a root cause.

13.4. Python-package 191
14.1 Algorithms

Refer to Features for understanding of important algorithms used in LightGBM.

14.2 Classes and Code Structure

14.2.1 Important Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>The entrance of application, including training and prediction logic</td>
</tr>
<tr>
<td>Bin</td>
<td>Data structure used for storing feature discrete values (converted from float values)</td>
</tr>
<tr>
<td>Boosting</td>
<td>Boosting interface (GBDT, DART, GOSS, etc.)</td>
</tr>
<tr>
<td>Config</td>
<td>Stores parameters and configurations</td>
</tr>
<tr>
<td>Dataset</td>
<td>Stores information of dataset</td>
</tr>
<tr>
<td>DatasetLoader</td>
<td>Used to construct dataset</td>
</tr>
<tr>
<td>FeatureGroup</td>
<td>Stores the data of feature, could be multiple features</td>
</tr>
<tr>
<td>Metric</td>
<td>Evaluation metrics</td>
</tr>
<tr>
<td>Network</td>
<td>Network interfaces and communication algorithms</td>
</tr>
<tr>
<td>ObjectiveFunction</td>
<td>Objective functions used to train</td>
</tr>
<tr>
<td>Tree</td>
<td>Stores information of tree model</td>
</tr>
<tr>
<td>TreeLearner</td>
<td>Used to learn trees</td>
</tr>
</tbody>
</table>
## 14.2.2 Code Structure

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>./include</td>
<td>Header files</td>
</tr>
<tr>
<td>./include/utils</td>
<td>Some common functions</td>
</tr>
<tr>
<td>./src/application</td>
<td>Implementations of training and prediction logic</td>
</tr>
<tr>
<td>./src/boosting</td>
<td>Implementations of Boosting</td>
</tr>
<tr>
<td>./src/io</td>
<td>Implementations of IO related classes, including Bin, Config, Dataset, DatasetLoader, Feature and Tree</td>
</tr>
<tr>
<td>./src/metric</td>
<td>Implementations of metrics</td>
</tr>
<tr>
<td>./src/network</td>
<td>Implementations of network functions</td>
</tr>
<tr>
<td>./src/objective</td>
<td>Implementations of objective functions</td>
</tr>
<tr>
<td>./src/treelearner</td>
<td>Implementations of tree learners</td>
</tr>
</tbody>
</table>

## 14.2.3 Documents API

Refer to docs README.

## 14.3 C API

Refer to C API or the comments in c_api.h file, from which the documentation is generated.

## 14.4 High Level Language Package

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

## 14.5 Questions

Refer to FAQ.

Also feel free to open issues if you met problems.
15.1 How It Works?

In LightGBM, the main computation cost during training is building the feature histograms. We use an efficient algorithm on GPU to accelerate this process. The implementation is highly modular, and works for all learning tasks (classification, ranking, regression, etc). GPU acceleration also works in distributed learning settings. GPU algorithm implementation is based on OpenCL and can work with a wide range of GPUs.

15.2 Supported Hardware

We target AMD Graphics Core Next (GCN) architecture and NVIDIA Maxwell and Pascal architectures. Most AMD GPUs released after 2012 and NVIDIA GPUs released after 2014 should be supported. We have tested the GPU implementation on the following GPUs:

- AMD RX 480 with AMDGPU-pro driver 16.60 on Ubuntu 16.10
- AMD R9 280X (aka Radeon HD 7970) with fglrx driver 15.302.2301 on Ubuntu 16.10
- NVIDIA GTX 1080 with driver 375.39 and CUDA 8.0 on Ubuntu 16.10
- NVIDIA Titan X (Pascal) with driver 367.48 and CUDA 8.0 on Ubuntu 16.04
- NVIDIA Tesla M40 with driver 375.39 and CUDA 7.5 on Ubuntu 16.04

Using the following hardware is discouraged:

- NVIDIA Kepler (K80, K40, K20, most GeForce GTX 700 series GPUs) or earlier NVIDIA GPUs. They don’t support hardware atomic operations in local memory space and thus histogram construction will be slow.
- AMD VLIW4-based GPUs, including Radeon HD 6xxx series and earlier GPUs. These GPUs have been discontinued for years and are rarely seen nowadays.

15.3 How to Achieve Good Speedup on GPU

1. You want to run a few datasets that we have verified with good speedup (including Higgs, epsilon, Bosch, etc) to ensure your setup is correct. If you have multiple GPUs, make sure to set `gpu_platform_id` and `gpu_device_id` to use the desired GPU. Also make sure your system is idle (especially when using a shared computer) to get accuracy performance measurements.

2. GPU works best on large scale and dense datasets. If dataset is too small, computing it on GPU is inefficient as the data transfer overhead can be significant. If you have categorical features, use the `categorical_column` option and input them into LightGBM directly; do not convert them into one-hot variables.
3. To get good speedup with GPU, it is suggested to use a smaller number of bins. Setting max_bin=63 is recommended, as it usually does not noticeably affect training accuracy on large datasets, but GPU training can be significantly faster than using the default bin size of 255. For some dataset, even using 15 bins is enough (max_bin=15); using 15 bins will maximize GPU performance. Make sure to check the run log and verify that the desired number of bins is used.

4. Try to use single precision training (gpu_use_dp=false) when possible, because most GPUs (especially NVIDIA consumer GPUs) have poor double-precision performance.

15.4 Performance Comparison

We evaluate the training performance of GPU acceleration on the following datasets:

<table>
<thead>
<tr>
<th>Data</th>
<th>Task</th>
<th>Link</th>
<th>#Examples</th>
<th>#Features</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higgs</td>
<td>Binary classification</td>
<td>link1</td>
<td>10,500,000</td>
<td>28</td>
<td>use last 500,000 samples as test set</td>
</tr>
<tr>
<td>Epsilon</td>
<td>Binary classification</td>
<td>link2</td>
<td>400,000</td>
<td>2,000</td>
<td>use the provided test set</td>
</tr>
<tr>
<td>Bosch</td>
<td>Binary classification</td>
<td>link3</td>
<td>1,000,000</td>
<td>968</td>
<td>use the provided test set</td>
</tr>
<tr>
<td>Yahoo LTR</td>
<td>Learning to rank</td>
<td>link4</td>
<td>473,134</td>
<td>700</td>
<td>set1.train as train, set1.test as test</td>
</tr>
<tr>
<td>MS LTR</td>
<td>Learning to rank</td>
<td>link5</td>
<td>2,270,296</td>
<td>137</td>
<td>{S1,S2,S3} as train set, {S5} as test set</td>
</tr>
<tr>
<td>Expo</td>
<td>Binary classification (Categorical)</td>
<td>link6</td>
<td>11,000,000</td>
<td>700</td>
<td>use last 1,000,000 as test set</td>
</tr>
</tbody>
</table>

We used the following hardware to evaluate the performance of LightGBM GPU training. Our CPU reference is a high-end dual socket Haswell-EP Xeon server with 28 cores; GPUs include a budget GPU (RX 480) and a mainstream (GTX 1080) GPU installed on the same server. It is worth mentioning that the GPUs used are not the best GPUs in the market; if you are using a better GPU (like AMD RX 580, NVIDIA GTX 1080 Ti, Titan X Pascal, Titan Xp, Tesla P100, etc), you are likely to get a better speedup.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Peak FLOPS</th>
<th>Peak Memory BW</th>
<th>Cost (MSRP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Radeon RX 480</td>
<td>5,161 GFLOPS</td>
<td>256 GB/s</td>
<td>$199</td>
</tr>
<tr>
<td>NVIDIA GTX 1080</td>
<td>8,228 GFLOPS</td>
<td>320 GB/s</td>
<td>$499</td>
</tr>
<tr>
<td>2x Xeon E5-2683x3 (28 cores)</td>
<td>1,792 GFLOPS</td>
<td>133 GB/s</td>
<td>$3,692</td>
</tr>
</tbody>
</table>

During benchmarking on CPU we used only 28 physical cores of the CPU, and did not use hyper-threading cores, because we found that using too many threads actually makes performance worse. The following shows the training configuration we used:

```python
max_bin = 63
num_leaves = 255
num_iterations = 500
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
device = gpu
```

(continues on next page)
We use the configuration shown above, except for the Bosch dataset, we use a smaller `learning_rate=0.015` and set `min_sum_hessian_in_leaf=5`. For all GPU training we vary the max number of bins (255, 63 and 15). The GPU implementation is from commit `0bb4a82` of LightGBM, when the GPU support was just merged in.

The following table lists the accuracy on test set that CPU and GPU learner can achieve after 500 iterations. GPU with the same number of bins can achieve a similar level of accuracy as on the CPU, despite using single precision arithmetic. For most datasets, using 63 bins is sufficient.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CPU 255 bins</th>
<th>CPU 63 bins</th>
<th>CPU 15 bins</th>
<th>GPU 255 bins</th>
<th>GPU 63 bins</th>
<th>GPU 15 bins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higgs AUC</td>
<td>0.845612</td>
<td>0.845239</td>
<td>0.841066</td>
<td>0.845612</td>
<td>0.845209</td>
<td>0.840748</td>
</tr>
<tr>
<td>Epsilon AUC</td>
<td>0.950243</td>
<td>0.949952</td>
<td>0.948365</td>
<td>0.950057</td>
<td>0.949876</td>
<td>0.948365</td>
</tr>
<tr>
<td>Yahoo-LTR NDCG</td>
<td>0.730824</td>
<td>0.730165</td>
<td>0.729647</td>
<td>0.730936</td>
<td>0.732257</td>
<td>0.73114</td>
</tr>
<tr>
<td>Yahoo-LTR NDCG</td>
<td>0.738687</td>
<td>0.737243</td>
<td>0.736445</td>
<td>0.73698</td>
<td>0.739474</td>
<td>0.735868</td>
</tr>
<tr>
<td>Yahoo-LTR NDCG</td>
<td>0.756609</td>
<td>0.755729</td>
<td>0.754607</td>
<td>0.756206</td>
<td>0.757007</td>
<td>0.754203</td>
</tr>
<tr>
<td>Yahoo-LTR NDCG</td>
<td>0.79655</td>
<td>0.795827</td>
<td>0.795273</td>
<td>0.795894</td>
<td>0.797302</td>
<td>0.795584</td>
</tr>
<tr>
<td>Expo AUC</td>
<td>0.776217</td>
<td>0.771566</td>
<td>0.743329</td>
<td>0.776285</td>
<td>0.77098</td>
<td>0.744078</td>
</tr>
<tr>
<td>MS-LTR NDCG</td>
<td>0.521265</td>
<td>0.521392</td>
<td>0.518653</td>
<td>0.521789</td>
<td>0.522163</td>
<td>0.516388</td>
</tr>
<tr>
<td>MS-LTR NDCG</td>
<td>0.503153</td>
<td>0.505753</td>
<td>0.501697</td>
<td>0.503886</td>
<td>0.504089</td>
<td>0.501691</td>
</tr>
<tr>
<td>MS-LTR NDCG</td>
<td>0.509236</td>
<td>0.510391</td>
<td>0.507193</td>
<td>0.509861</td>
<td>0.510095</td>
<td>0.50663</td>
</tr>
<tr>
<td>MS-LTR NDCG</td>
<td>0.527835</td>
<td>0.527304</td>
<td>0.524603</td>
<td>0.528009</td>
<td>0.527059</td>
<td>0.524722</td>
</tr>
<tr>
<td>Bosch AUC</td>
<td>0.718115</td>
<td>0.721791</td>
<td>0.716677</td>
<td>0.717184</td>
<td>0.724761</td>
<td>0.717005</td>
</tr>
</tbody>
</table>

We record the wall clock time after 500 iterations, as shown in the figure below:

When using a GPU, it is advisable to use a bin size of 63 rather than 255, because it can speed up training significantly without noticeably affecting accuracy. On CPU, using a smaller bin size only marginally improves performance, sometimes even slows down training, like in Higgs (we can reproduce the same slowdown on two different machines, with different GCC versions). We found that GPU can achieve impressive acceleration on large and dense datasets like...
Higgs and Epsilon. Even on smaller and sparse datasets, a budget GPU can still compete and be faster than a 28-core Haswell server.

### 15.5 Memory Usage

The next table shows GPU memory usage reported by `nvidia-smi` during training with 63 bins. We can see that even the largest dataset just uses about 1 GB of GPU memory, indicating that our GPU implementation can scale to huge datasets over 10x larger than Bosch or Epsilon. Also, we can observe that generally a larger dataset (using more GPU memory, like Epsilon or Bosch) has better speedup, because the overhead of invoking GPU functions becomes significant when the dataset is small.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Higgs</th>
<th>Epsilon</th>
<th>Bosch</th>
<th>MS-LTR</th>
<th>Expo</th>
<th>Yahoo-LTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Memory Usage (MB)</td>
<td>611</td>
<td>901</td>
<td>1067</td>
<td>413</td>
<td>405</td>
<td>291</td>
</tr>
</tbody>
</table>

### 15.6 Further Reading

You can find more details about the GPU algorithm and benchmarks in the following article:

CHAPTER
SIXTEEN

GPU SDK CORRESPONDENCE AND DEVICE TARGETING TABLE

16.1 GPU Targets Table

OpenCL is a universal massively parallel programming framework that targets to multiple backends (GPU, CPU, FPGA, etc). Basically, to use a device from a vendor, you have to install drivers from that specific vendor. Intel’s and AMD’s OpenCL runtime also include x86 CPU target support. NVIDIA’s OpenCL runtime only supports NVIDIA GPU (no CPU support). In general, OpenCL CPU backends are quite slow, and should be used for testing and debugging only.

You can find below a table of correspondence:

<table>
<thead>
<tr>
<th>SDK</th>
<th>CPU Intel/AMD</th>
<th>GPU Intel</th>
<th>GPU AMD</th>
<th>GPU NVIDIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel SDK for OpenCL</td>
<td>Supported</td>
<td>Supported</td>
<td>Not Supported</td>
<td>Not Supported</td>
</tr>
<tr>
<td>AMD APP SDK *</td>
<td>Supported</td>
<td>Not Supported</td>
<td>Supported</td>
<td>Not Supported</td>
</tr>
<tr>
<td>NVIDIA CUDA Toolkit</td>
<td>Not Supported</td>
<td>Not Supported</td>
<td>Not Supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>

Legend:

* AMD APP SDK is deprecated. On Windows, OpenCL is included in AMD graphics driver. On Linux, newer generation AMD cards are supported by the ROCm driver. You can download an archived copy of AMD APP SDK from our GitHub repo (for Linux and for Windows).

16.2 Query OpenCL Devices in Your System

Your system might have multiple GPUs from different vendors ("platforms") installed. Setting up LightGBM GPU device requires two parameters: OpenCL Platform ID (gpu_platform_id) and OpenCL Device ID (gpu_device_id). Generally speaking, each vendor provides an OpenCL platform, and devices from the same vendor have different device IDs under that platform. For example, if your system has an Intel integrated GPU and two discrete GPUs from AMD, you will have two OpenCL platforms (with gpu_platform_id=0 and gpu_platform_id=1). If the platform 0 is Intel, it has one device (gpu_device_id=0) representing the Intel GPU; if the platform 1 is AMD, it has two devices (gpu_device_id=0, gpu_device_id=1) representing the two AMD GPUs. If you have a discrete GPU by AMD/NVIDIA and an integrated GPU by Intel, make sure to select the correct gpu_platform_id to use the discrete GPU as it usually provides better performance.

On Windows, OpenCL devices can be queried using GPUCapsViewer, under the OpenCL tab. Note that the platform and device IDs reported by this utility start from 1. So you should minus the reported IDs by 1.

On Linux, OpenCL devices can be listed using the clinfo command. On Ubuntu, you can install clinfo by executing sudo apt-get install clinfo.
16.3 Examples

We provide test R code below, but you can use the language of your choice with the examples of your choices:

```r
library(lightgbm)
data(agaricus.train, package = "lightgbm")
train <- agaricus.train
data[, 1] <- 1:6513
dtrain <- lgb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "lightgbm")
test <- agaricus.test
data(test$data, label = test$label)
dtest <- lgb.Dataset.create.valid(dtrain, test$data, label = test$label)
valids <- list(test = dtest)

params <- list(objective = "regression",
               metric = "rmse",
               device = "gpu",
               gpu_platform_id = 0,
               gpu_device_id = 0,
               nthread = 1,
               boost_from_average = FALSE,
               num_tree_per_iteration = 10,
               max_bin = 32)
model <- lgb.train(params,
                   dtrain,
                   2,
                   valids,
                   min_data = 1,
                   learning_rate = 1,
                   early_stopping_rounds = 10)
```

Make sure you list the OpenCL devices in your system and set `gpu_platform_id` and `gpu_device_id` correctly. In the following examples, our system has 1 GPU platform (`gpu_platform_id = 0`) from AMD APP SDK. The first device `gpu_device_id = 0` is a GPU device (AMD Oland), and the second device `gpu_device_id = 1` is the x86 CPU backend.

Example of using GPU (`gpu_platform_id = 0` and `gpu_device_id = 0` in our system):

```r
params <- list(objective = "regression",
               metric = "rmse",
               device = "gpu",
               gpu_platform_id = 0,
               gpu_device_id = 0,
               nthread = 1,
               boost_from_average = FALSE,
               num_tree_per_iteration = 10,
               max_bin = 32)
model <- lgb.train(params,
                   dtrain,
                   2,
                   valids,
                   min_data = 1,
                   learning_rate = 1,
                   early_stopping_rounds = 10)
```

[LightGBM] [Info] This is the GPU trainer!!
[LightGBM] [Info] Total Bins 232
[LightGBM] [Info] Number of data: 6513, number of used features: 116
Running on OpenCL CPU backend devices is in generally slow, and we observe crashes on some Windows and macOS systems. Make sure you check the Using GPU Device line in the log and it is not using a CPU. The above log shows that we are using Oland GPU from AMD and not CPU.

Example of using CPU (gpu_platform_id = 0, gpu_device_id = 1). The GPU device reported is Intel(R) Core(TM) i7-4600U CPU, so it is using the CPU backend rather than a real GPU.

```r
> params <- list(objective = "regression",
+     metric = "rmse",
+     device = "gpu",
+     gpu_platform_id = 0,
+     gpu_device_id = 1,
+     nthread = 1,
+     boost_from_average = FALSE,
+     num_tree_per_iteration = 10,
+     max_bin = 32)
> model <- lgb.train(params,
+     dtrain,
+     2,
+     valids,
+     min_data = 1,
+     learning_rate = 1,
+     early_stopping_rounds = 10)
[LightGBM] [Info] This is the GPU trainer!!
[LightGBM] [Info] Total Bins 232
[LightGBM] [Info] Number of data: 6513, number of used features: 116
[LightGBM] [Info] Using requested OpenCL platform 0 device 1
[LightGBM] [Info] Using GPU Device: Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz, Vendor:
+     GenuineIntel
[LightGBM] [Info] Compiling OpenCL Kernel with 16 bins...
[LightGBM] [Info] GPU programs have been built
[LightGBM] [Info] Size of histogram bin entry: 12
[LightGBM] [Info] 40 dense feature groups (0.12 MB) transferred to GPU in 0.004540
+     secs. 76 sparse feature groups.
[LightGBM] [Info] No further splits with positive gain, best gain: -inf
[LightGBM] [Info] Trained a tree with leaves=16 and depth=8
[1]:  test's rmse:1.10643e-17
[LightGBM] [Info] No further splits with positive gain, best gain: -inf
[LightGBM] [Info] Trained a tree with leaves=7 and depth=5
[2]:  test's rmse:0
```

Known issues:

- Using a bad combination of `gpu_platform_id` and `gpu_device_id` can potentially lead to a crash due to OpenCL driver issues on some machines (you will lose your entire session content). Beware of it.
• On some systems, if you have integrated graphics card (Intel HD Graphics) and a dedicated graphics card (AMD, NVIDIA), the dedicated graphics card will automatically override the integrated graphics card. The workaround is to disable your dedicated graphics card to be able to use your integrated graphics card.
GPU WINDOWS COMPILATION

This guide is for the MinGW build.

For the MSVC (Visual Studio) build with GPU, please refer to Installation Guide. (We recommend you to use this since it is much easier).

### 17.1 Install LightGBM GPU version in Windows (CLI / R / Python), using MinGW/gcc

This is for a vanilla installation of Boost, including full compilation steps from source without precompiled libraries.

Installation steps (depends on what you are going to do):

- Install the appropriate OpenCL SDK
- Install MinGW
- Install Boost
- Install Git
- Install CMake
- Create LightGBM binaries
- Debugging LightGBM in CLI (if GPU is crashing or any other crash reason)

If you wish to use another compiler like Visual Studio C++ compiler, you need to adapt the steps to your needs.

For this compilation tutorial, we are using AMD SDK for our OpenCL steps. However, you are free to use any OpenCL SDK you want, you just need to adjust the PATH correctly.

You will also need administrator rights. This will not work without them.

At the end, you can restore your original PATH.
17.1.1 Modifying PATH (for newbies)

To modify PATH, just follow the pictures after going to the Control Panel:

Then, go to Advanced > Environment Variables...:
Under `System variables`, the variable `Path`: 
Antivirus Performance Impact

Does not apply to you if you do not use a third-party antivirus nor the default preinstalled antivirus on Windows. **Windows Defender or any other antivirus will have a significant impact on the speed you will be able to perform the steps.** It is recommended to **turn them off temporarily** until you finished with building and setting up everything, then turn them back on, if you are using them.
17.1.2 OpenCL SDK Installation

Installing the appropriate OpenCL SDK requires you to download the correct vendor source SDK. You need to know what you are going to use LightGBM!

- For running on Intel, get Intel SDK for OpenCL (**NOT RECOMMENDED**).
- For running on AMD, get AMD APP SDK (downloads for Linux and for Windows). You may want to replace the `OpenCL.dll` from the GPU driver package with the one from the SDK, if the one shipped with the driver lacks some functions.
- For running on NVIDIA, get CUDA Toolkit.
- Or you can try to use Khronos official OpenCL headers, the CMake module would automatically find the OpenCL library used in your system, though the result may be not portable.

Further reading and correspondence table (especially if you intend to use cross-platform devices, like Intel CPU with AMD APP SDK): GPU SDK Correspondence and Device Targeting Table.

**Warning**: using Intel OpenCL is not recommended and may crash your machine due to being non compliant to OpenCL standards. If your objective is to use LightGBM + OpenCL on CPU, please use AMD APP SDK instead (it can run also on Intel CPUs without any issues).

17.1.3 MinGW Correct Compiler Selection

If you are expecting to use LightGBM without R, you need to install MinGW. Installing MinGW is straightforward, download this.

Make sure you are using the x86_64 architecture, and do not modify anything else. You may choose a version other than the most recent one if you need a previous MinGW version.
Then, add to your PATH the following (to adjust to your MinGW version):

```
C:\Program Files\mingw-w64\x86_64-5.3.0-posix-seh-rt_v4-rev0\mingw64\bin
```

**Warning**: R users (even if you do not want LightGBM for R)

If you have RTools and MinGW installed, and wish to use LightGBM in R, get rid of MinGW from PATH (to keep: `c:\Rtools\bin; c:\Rtools\mingw_32\bin` for 32-bit R installation, `c:\Rtools\bin; c:\Rtools\mingw_64\bin` for 64-bit R installation).

You can check which MinGW version you are using by running the following in a command prompt: `gcc -v`:

To check whether you need 32-bit or 64-bit MinGW for R, install LightGBM as usual and check for the following:
* installing *source* package 'lightgbm' ...
  ** libs
c:/Rtools/mingw_64/bin/g++

If it says mingw_64 then you need the 64-bit version (PATH with c:/Rtools\bin;c:/Rtools\mingw_64\bin), otherwise you need the 32-bit version (c:/Rtools\bin;c:/Rtools\mingw_32\bin), the latter being a very rare and untested case.

NOTE: If you are using Rtools 4.0 or later, the path will have mingw64 instead of mingw_64 (PATH with C:\rtools\mingw64\bin), and mingw32 instead of mingw_32 (C:\rtools\mingw32\bin). The 32-bit version remains an unsupported solution under Rtools 4.0.

### 17.1.4 Download the prebuilt Boost

Download Prebuilt Boost x86_64 or Prebuilt Boost i686 and unpack them with 7zip, alternatively you can build Boost from source.

### 17.1.5 Boost Compilation

Installing Boost requires to download Boost and to install it. It takes about 10 minutes to several hours depending on your CPU speed and network speed.

We will assume an installation in C:\boost and a general installation (like in Unix variants: without versioning and without type tags).

There is one mandatory step to check the compiler:

- **Warning**: if you want the R installation: If you have already MinGW in your PATH variable, get rid of it (you will link to the wrong compiler otherwise).

- **Warning**: if you want the CLI installation: If you have already Rtools in your PATH variable, get rid of it (you will link to the wrong compiler otherwise).

- R installation must have Rtools in PATH

- CLI / Python installation must have MinGW (not Rtools) in PATH

In addition, assuming you are going to use C:\boost for the folder path, you should add now already the following to PATH: C:\boost\boost\build\bin, C:\boost\boost\build\include\boost. Adjust C:\boost if you install it elsewhere.

We can now start downloading and compiling the required Boost libraries:

- Download Boost (for example, the filename for 1.63.0 version is boost_1_63_0.zip)
- Extract the archive to C:\boost
- Open a command prompt, and run

```
cd C:\boost\boost_1_63_0\tools\build
bootstrap.bat gcc
b2 install --prefix="C:\boost\boost\build" toolset=gcc
cd C:\boost\boost_1_63_0
```

To build the Boost libraries, you have two choices for command prompt:

- If you have only one single core, you can use the default
If you want to do a multithreaded library building (faster), add `-j N` by replacing N by the number of cores/threads you have. For instance, for 2 cores, you would do

```
b2 install --build_dir="C:\boost\boost-build" --prefix="C:\boost\boost-build" --toolset=gcc --with=filesystem,system threading=multi --layout=system release -j 2
```

Ignore all the errors popping up, like Python, etc., they do not matter for us.

Your folder should look like this at the end (not fully detailed):

```
- C
  |--- boost
  |    |--- boost_1_63_0
  |    |    |--- some folders and files
  |    |    |--- boost-build
  |    |    |    |--- bin
  |    |    |    |--- include
  |    |    |    |--- boost
  |    |    |--- lib
  |    |--- share
```

This is what you should (approximately) get at the end of Boost compilation:

```
bin/0x116type_traits/build/gcc-x86_64/debug/link/static/threading=multi/libboost_type_traits-msys3-mt-d-1_63.a 1 file(s) copied.
common/addir bin/0x116type_traits/build/gcc-x86_64/debug/link/static/threading=multi/libboost_type_traits-msys3-mt-d-1_63.a 1 file(s) copied.
common/addir bin/0x116type_traits/build/gcc-x86_64/debug/link/static/threading=multi/libboost_type_traits-msys3-mt-d-1_63.a 1 file(s) copied.
```

If you are getting an error:

- Wipe your Boost directory
- Close the command prompt
- Make sure you added C:\boost\boost-build\bin, C:\boost\boost-build\include\boost to your PATH (adjust accordingly if you use another folder)
- Do the Boost compilation steps again (extract => command prompt => cd => bootstrap => b2 => cd => b2)
17.1.6 Git Installation

Installing Git for Windows is straightforward, use the following link.

Now, we can fetch LightGBM repository for GitHub. Run Git Bash and the following command:

```
cd C:/
mkdir github_repos
cd github_repos
git clone --recursive https://github.com/microsoft/LightGBM
```

Your LightGBM repository copy should now be under `C:\github_repos\LightGBM`. You are free to use any folder you want, but you have to adapt.

Keep Git Bash open.

17.1.7 CMake Installation, Configuration, Generation

CLI / Python users only

Installing CMake requires one download first and then a lot of configuration for LightGBM:

Binary distributions:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows win64 x64</td>
<td>cmake-3.8.0-win64-x64.msi</td>
</tr>
</tbody>
</table>

- Download CMake (3.8 or higher)
• Install CMake
• Run cmake-gui
• Select the folder where you put LightGBM for *Where is the source code, default using our steps would be* C:/github_repos/LightGBM
• Copy the folder name, and add /build for “Where to build the binaries”, default using our steps would be C:/github_repos/LightGBM/build
• Click *Configure*

• Lookup for *USE_GPU* and check the checkbox
• Click **Configure**

You should get (approximately) the following after clicking Configure:
Looking for CL_VERSION_2_0
Looking for CL_VERSION_2_0 - found
Found OpenCL: C:/Windows/System32/OpenCL.dll (found version "2.0")
OpenCL include directory:C:/Program Files (x86)/AMD APP SDK/3.0/include
Boost version: 1.63.0
Found the following Boost libraries:
  filesystem
  system
Configuring done

- Click Generate to get the following message:

  Generating done

This is straightforward, as CMake is providing a large help into locating the correct elements.
17.1.8 LightGBM Compilation (CLI: final step)

Installation in CLI

CLI / Python users

Creating LightGBM libraries is very simple as all the important and hard steps were done before. You can do everything in the Git Bash console you left open:

- If you closed Git Bash console previously, run this to get back to the build folder:
  
  ```bash
  cd C:/github_repos/LightGBM/build
  ```

- If you did not close the Git Bash console previously, run this to get to the build folder:
  
  ```bash
  cd LightGBM/build
  ```

- Setup MinGW as `make` using
  
  ```bash
  alias make='mingw32-make'
  ```
  
  otherwise, beware error and name clash!

- In Git Bash, run `make` and see LightGBM being installing!

If everything was done correctly, you now compiled CLI LightGBM with GPU support!
Testing in CLI

You can now test LightGBM directly in CLI in a command prompt (not Git Bash):

```
cd C:/github_repos/LightGBM/examples/binary_classification
"../../../lightgbm.exe" config=train.conf data=binary.train valid=binary.test _
→ objective=binary device=gpu
```

Congratulations for reaching this stage!

To learn how to target a correct CPU or GPU for training, please see: GPU SDK Correspondence and Device Targeting Table.

17.1.9 Debugging LightGBM Crashes in CLI

Now that you compiled LightGBM, you try it... and you always see a segmentation fault or an undocumented crash with GPU support:

```
[New Thread 105220.0x194901]
[New Thread 105220.0x1a71c1]
[New Thread 105220.0x19a241]
[New Thread 105220.0x1f0b01]
[Thread 105220.0xf4b08 exited with code 0]
[LightGBM] [Info] Loading weights...
[New Thread 105220.0x197881]
[Thread 105220.0x19988 exited with code 0]
[New Thread 105220.0x1a8fb1]
[Thread 105220.0x1a8fc exited with code 0]
[LightGBM] [Info] Loading weights...
[New Thread 105220.0x1a90e1]
[Thread 105220.0x1a90c exited with code 0]
[LightGBM] [Info] Finished loading data in 1.011408 seconds
[LightGBM] [Info] Number of positive: 3715, number of negative: 3284
[LightGBM] [Info] This is the GPU trainer!
[LightGBM] [Info] Total Bins 6142
[LightGBM] [Info] Number of data: 7000, number of used features: 28
[New Thread 105220.0x1a62c1]
[LightGBM] [Info] Compiling OpenCL Kernel with 256 bins...
```

Please check if you are using the right device (Using GPU device: ...). You can find a list of your OpenCL devices using GPUCapsViewer, and make sure you are using a discrete (AMD/NVIDIA) GPU if you have both integrated (Intel) and discrete GPUs installed. Also, try to set `gpu_device_id = 0` and `gpu_platform_id = 0` or `gpu_device_id = -1` and `gpu_platform_id = -1` to use the first platform and device or the default platform and device. If it still does not work, then you should follow all the steps below.
You will have to redo the compilation steps for LightGBM to add debugging mode. This involves:

- Deleting `C:/github_repos/LightGBM/build` folder
- Deleting `lightgbm.exe`, `lib_lightgbm.dll`, and `lib_lightgbm.dll.a` files

Once you removed the file, go into CMake, and follow the usual steps. Before clicking “Generate”, click on “Add Entry”:
In addition, click on Configure and Generate:
And then, follow the regular LightGBM CLI installation from there.

Once you have installed LightGBM CLI, assuming your LightGBM is in `C:\github_repos\LightGBM`, open a command prompt and run the following:

```bash
gdb --args "../../lightgbm.exe" config=train.conf data=binary.train valid=binary.test objective=binary device=gpu
```

Type `run` and press the Enter key.
LightGBM, Release 3.2.0.99

You will probably get something similar to this:

```
[LightGBM] [Info] This is the GPU trainer!!
[LightGBM] [Info] Total Bins 6143
[LightGBM] [Info] Number of data: 7000, number of used features: 28
[New Thread 105220.0x1a62c]
[LightGBM] [Info] Compiling OpenCL Kernel with 256 bins...
```

Program received signal SIGSEGV, Segmentation fault.
0x00007ffbb37c11f1 in strlen () from C:\Windows\system32\msvcrt.dll
(gdb)

There, write `backtrace` and press the Enter key as many times as gdb requests two choices:

```
Program received signal SIGSEGV, Segmentation fault.
0x00007ffbb37c11f1 in strlen () from C:\Windows\system32\msvcrt.dll (gdb) backtrace
#0 0x000000000048be5 in std::char_traits<char>::length (__s=0x0) at C:/PROGRA~1/MINGW-~1/X86_64~1.0-P/mingw64/x86_64-w64-mingw32/include/c++/bits/char_traits.h:267
#1 std::operator+(char, std::char_traits<char>, std::allocator<char>) > (_rhs="\", _lhs=0x0) at C:/PROGRA~1/MINGW-~1/X86_64~1.0-P/mingw64/x86_64-w64-mingw32/include/c++/bits/basic_string.tcc:1157
#2 boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#3 0x000000000048bee3 in std::operator+(char, std::char_traits<char>, std::allocator<char>) > (_rhs="\", _lhs=0x0) at C:/PROGRA~1/MINGW-~1/X86_64~1.0-P/mingw64/x86_64-w64-mingw32/include/c++/bits/basic_string.tcc:1157
#4 0x000000000048bee3 in boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#5 boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#6 0x000000000048bee3 in boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#7 boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#8 boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#9 boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#10 boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#11 boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
#12 boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/include/boost/compute/detail/path.hpp:38
```

(continues on next page)
Right-click the command prompt, click “Mark”, and select all the text from the first line (with the command prompt containing gdb) to the last line printed, containing all the log, such as:

```
C:\LightGBM\examples\binary_classification>gdb --args "../../lightgbm.exe"
... ...
```

17.1. Install LightGBM GPU version in Windows (CLI / R / Python), using MinGW/gcc
And open an issue in GitHub here with that log.
It is recommended to use `-O3 -mtune=native` to achieve maximum speed during LightGBM training.

Using Intel Ivy Bridge CPU on 1M x 1K Bosch dataset, the performance increases as follow:

<table>
<thead>
<tr>
<th>Compilation Flag</th>
<th>Performance Index</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-O2 -mtune=core2</code></td>
<td>100.00%</td>
</tr>
<tr>
<td><code>-O2 -mtune=native</code></td>
<td>100.90%</td>
</tr>
<tr>
<td><code>-O3 -mtune=native</code></td>
<td>102.78%</td>
</tr>
<tr>
<td><code>-O3 -ffast-math -mtune=native</code></td>
<td>100.64%</td>
</tr>
</tbody>
</table>

You can find more details on the experimentation below:

- Laurae++/Benchmarks
- Laurae2/gbt_benchmarks
- Laurae’s Benchmark Master Data (Interactive)
- Kaggle Paris Meetup #12 Slides

Some explanatory pictures:
Chapter 18. Recommendations When Using gcc
Documentation for LightGBM is generated using Sphinx and Breathe, which works on top of Doxygen output.

List of parameters and their descriptions in Parameters.rst is generated automatically from comments in config file by this script.

After each commit on master, documentation is updated and published to Read the Docs.

### 19.1 Build

You can build the documentation locally. Just install Doxygen and run in docs folder

```sh
pip install -r requirements.txt
make html
```

Unfortunately, documentation for R code is built only on our site, and commands above will not build it for you locally. Consider using common R utilities for documentation generation, if you need it.

If you faced any problems with Doxygen installation or you simply do not need documentation for C code, it is possible to build the documentation without it:

```sh
pip install -r requirements_base.txt
export C_API=NO || set C_API=NO
make html
```
INDICES AND TABLES

• genindex
Symbols

__init__() (lightgbm.Booster method), 101
__init__() (lightgbm.CVBooster method), 110
__init__() (lightgbm.DaskLGBMClassifier method), 144
__init__() (lightgbm.DaskLGBMRegressor method), 156
__init__() (lightgbm.DaskLGBMModel method), 150
__init__() (lightgbm.Dataset method), 95
__init__() (lightgbm.LGBMClassifier method), 122
__init__() (lightgbm.LGBMModel method), 116
__init__() (lightgbm.LGBMRanker method), 137
__init__() (lightgbm.LGBMRegressor method), 130
add_features_from() (lightgbm.Dataset method), 97
add_valid() (lightgbm.Booster method), 102
attr() (lightgbm.Booster method), 102

A

best_iteration (lightgbm.CVBooster attribute), 110
best_iteration_() (lightgbm.DaskLGBMClassifier property), 146
best_iteration_() (lightgbm.DaskLGBMRegressor property), 153
best_iteration_() (lightgbm.DaskLGBMRanker property), 158
best_iteration_() (lightgbm.LGBMClassifier property), 125
best_iteration_() (lightgbm.LGBMModel property), 118
best_iteration_() (lightgbm.LGBMRanker property), 139
best_iteration_() (lightgbm.LGBMRegressor property), 133

B

booster_() (lightgbm.DaskLGBMClassifier property), 146
booster_() (lightgbm.DaskLGBMRanker property), 158
booster_() (lightgbm.DaskLGBMRegressor property), 153
booster_() (lightgbm.LGBMClassifier property), 125
booster_() (lightgbm.LGBMModel property), 118
booster_() (lightgbm.LGBMRanker property), 139
booster_() (lightgbm.LGBMRegressor property), 133
BoosterHandle (C type), 66
boosters (lightgbm.CVBooster attribute), 110

C

C_API_DTYPE_FLOAT32 (C macro), 65
C_API_DTYPE_FLOAT64 (C macro), 65
C_API_DTYPE_INT32 (C macro), 65
C_API_DTYPE_INT64 (C macro), 65
C_API_PREDICT_CONTRIB (C macro), 65
C_API_PREDICT_LEAF_INDEX (C macro), 65
C_API_PREDICT_NORMAL (C macro), 65
C_API_PREDICT_RAW_SCORE (C macro), 65
classes_() (lightgbm.DaskLGBMClassifier property), 146
classes_() (lightgbm.LGBMClassifier property), 125
get_params () (lightgbm.LGBMRegressor method), 135
get_ref_chain () (lightgbm.Dataset method), 98
get_split_value_histogram () (lightgbm.Booster method), 105
get_weight () (lightgbm.Dataset method), 98

LastErrorMsg (C function), 66
LGBM_BoosterAddValidData (C function), 66
LGBM_BoosterCalcNumPredict (C function), 66
LGBM_BoosterCreate (C function), 67
LGBM_BoosterCreateFromModelfile (C function), 67
LGBM_BoosterDumpModel (C function), 67
LGBM_BoosterFeatureImportance (C function), 67
LGBM_BoosterFree (C function), 68
LGBM_BoosterFreePredictSparse (C function), 68
LGBM_BoosterGetCurrentIteration (C function), 68
LGBM_BoosterGetEval (C function), 69
LGBM_BoosterGetEvalCounts (C function), 69
LGBM_BoosterGetEvalNames (C function), 69
LGBM_BoosterGetFeatureNames (C function), 69
LGBM_BoosterGetLeafValue (C function), 70
LGBM_BoosterGetLinear (C function), 70
LGBM_BoosterGetLowerBoundValue (C function), 70
LGBM_BoosterGetNumClasses (C function), 71
LGBM_BoosterGetNumFeature (C function), 71
LGBM_BoosterGetNumPredict (C function), 71
LGBM_BoosterGetPredict (C function), 71
LGBM_BoosterGetUpperBoundValue (C function), 71
LGBM_BoosterLoadModelFromString (C function), 72
LGBM_BoosterMerge (C function), 72
LGBM_BoosterNumberOfTotalModel (C function), 72
LGBM_BoosterNumModelPerIteration (C function), 72
LGBM_BoosterPredictForCSC (C function), 72
LGBM_BoosterPredictForCSR (C function), 73
LGBM_BoosterPredictForCSRSingleRow (C function), 74
LGBM_BoosterPredictForCSRSingleRowFast (C function), 76
LGBM_BoosterPredictForCSRSingleRowFastInit (C function), 76
LGBM_BoosterPredictForFile (C function), 77
LGBM_BoosterPredictForMat (C function), 78
LGBM_BoosterPredictForMats (C function), 79
LGBM_BoosterPredictForMatSingleRow (C function), 79
LGBM_BoosterPredictForMatSingleRowFast (C function), 80
LGBM_BoosterPredictForMatSingleRowFastInit (C function), 81
LGBM_BoosterPredictSparseOutput (C function), 82
LGBM_BoosterRefit (C function), 83
LGBM_BoosterResetParameter (C function), 83
LGBM_BoosterResetTrainingData (C function), 83
LGBM_BoosterRollbackOneIter (C function), 83
LGBM_BoosterSaveModel (C function), 83
LGBM_BoosterSaveModelToString (C function), 84
LGBM_BoosterSetLeafValue (C function), 84
LGBM_BoosterShuffleModels (C function), 84
LGBM_BoosterUpdateOneIter (C function), 85
LGBM_BoosterUpdateOneIterCustom (C function), 85
LGBM_BoosterDatasetAddFeaturesFrom (C function), 85
LGBM_BoosterDatasetCreateByReference (C function), 85
LGBM_BoosterDatasetCreateFromCSC (C function), 86
LGBM_BoosterDatasetCreateFromCSR (C function), 86
LGBM_BoosterDatasetCreateFromCSRFunc (C function), 87
LGBM_BoosterDatasetCreateFromFile (C function), 87
LGBM_BoosterDatasetCreateFromMat (C function), 88
LGBM_BoosterDatasetCreateFromMats (C function), 88
LGBM_BoosterDatasetCreateFromSampledColumn (C function), 88
LGBM_BoosterDatasetDumpText (C function), 89
LGBM_BoosterDatasetFree (C function), 89
LGBM_BoosterDatasetGetFeatureNames (C function), 90
LGBM_BoosterDatasetGetField (C function), 90
LGBM_BoosterDatasetGetNumData (C function), 90
LGBM_BoosterDatasetGetNumFeature (C function), 90
LGBM_BoosterDatasetGetNumFeatureNames (C function), 90
LGBM_BoosterDatasetPushRows (C function), 91
LGBM_BoosterDatasetPushRowsByCSR (C function), 91
LGBM_BoosterDatasetSaveBinary (C function), 92
LGBM_BoosterDatasetSetFeatureNames (C function), 92
LGBM_BoosterDatasetSetField (C function), 92
LGBM_BoosterDatasetUpdateParamChecking (C function), 93
LGBM_FastConfigFree (C function), 93
LGBM_GetLastError (C function), 93
LGBM_NetworkFree (C function), 93
LGBM_NetworkInit (C function), 93
LGBM_NetworkInitWithFunctions (C function), 93
LGBM_RegisterLogCallback (C function), 94
set_feature_name() (lightgbm.Dataset method), 99
set_field() (lightgbm.Dataset method), 99
set_group() (lightgbm.Dataset method), 99
set_init_score() (lightgbm.Dataset method), 100
set_label() (lightgbm.Dataset method), 100
set_network() (lightgbm.Booster method), 108
set_params() (lightgbm.DaskLGBMClassifier method), 150
set_params() (lightgbm.DaskLGBMRanker method), 161
set_params() (lightgbm.DaskLGBMRegressor method), 155
set_params() (lightgbm.LGBMClassifier method), 130
set_params() (lightgbm.LGBMModel method), 122
set_params() (lightgbm.LGBMRanker method), 143
set_params() (lightgbm.LGBMRegressor method), 136
set_reference() (lightgbm.Dataset method), 100
set_train_data_name() (lightgbm.Booster method), 108
set_weight() (lightgbm.Dataset method), 100
shuffle_models() (lightgbm.Booster method), 108
subset() (lightgbm.Dataset method), 100

T
THREAD_LOCAL (C macro), 66
to_local() (lightgbm.DaskLGBMClassifier method), 150
to_local() (lightgbm.DaskLGBMRanker method), 161
to_local() (lightgbm.DaskLGBMRegressor method), 155
train() (in module lightgbm), 111
trees_to_dataframe() (lightgbm.Booster method), 109

U
update() (lightgbm.Booster method), 109
upper_bound() (lightgbm.Booster method), 110