LightGBM

Mar 22, 2019

Contents:

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 and GPU learning.
- Capable of handling large-scale data.

For more details, please refer to Features.

CHAPTER 1

Installation Guide

Here is the guide for the build of LightGBM CLI version.

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

Also you can download artifacts of the latest successful build in master branch: .

Contents

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.
3. Go to `LightGBM-master/windows` folder.
4. Open `LightGBM.sln` file with **Visual Studio**, choose `Release` configuration and click `BUILD` -> `Build Solution (Ctrl+Shift+B)`.

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

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

From Command Line

1. Install Git for Windows, CMake (3.8 or higher) and 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 -DCMAKE_GENERATOR_PLATFORM=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 to use **Visual Studio** for its better multithreading efficiency in **Windows** for many-core systems (see FAQ 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:

```
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 (3.12 or higher):

```
brew install cmake
```

2. Install **OpenMP**:

```
brew install libomp
```

3. Run the following commands:

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

# For Mojave (10.14)
cmake \
  -DOpenMP_C_FLAGS="-Xpreprocessor -fopenmp -I$(brew --prefix libomp)/include" \
  -DOpenMP_C_LIB_NAMES="omp" \
  -DOpenMP_CXX_FLAGS="-Xpreprocessor -fopenmp -I$(brew --prefix libomp)/include" \
  -DOpenMP_CXX_LIB_NAMES="omp" \
  -DOpenMP_omp_LIBRARY=$(brew --prefix libomp)/lib/libomp.dylib \
  ..

# For High Sierra or earlier (<= 10.13)
cmake ..

make -j4
```

1.3.2 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 ..
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. However, you can build the LightGBM without OpenMP support, but it is **strongly not recommended**.

1.5.1 Windows

On Windows 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.
3. Go to `LightGBM-master/windows` folder.
4. Open `LightGBM.sln` file with **Visual Studio**.
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:

```
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -DCMAKE_GENERATOR_PLATFORM=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:

```
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -G "MinGW Makefiles" -DUSE_OPENMP=OFF ..
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" -DUSE_OPENMP=OFF ..` one more time if you encounter the `sh.exe was found in your PATH` error.

1.5.2 Linux

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

1. 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
```

Note: glibc >= 2.14 is required.

1.5.3 macOS

On macOS 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 (3.12 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 parallel learning application with high performance communication, you can build the LightGBM with MPI support.

1.6.1 Windows

On Windows 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 `msmpisetup.exe` are needed.
2. Install Visual Studio (2015 or newer).
3. Download zip archive and unzip it.
4. Go to `LightGBM-master/windows` folder.
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 `msmpisetup.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:

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

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

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

1.6.2 Linux

On Linux 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 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 (3.12 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

# For Mojave (10.14)
cmake \
  -DUSE_MPI=ON \
  -DOpenMP_C_FLAGS="-Xpreprocessor -fopenmp -I$(brew --prefix libomp)/include" \
  -DOpenMP_C_LIB_NAMES="omp" \
  -DOpenMP_CXX_FLAGS="-Xpreprocessor -fopenmp -I$(brew --prefix libomp)/include" \
  -DOpenMP_CXX_LIB_NAMES="omp" \
  -DOpenMP_omp_LIBRARY=$(brew --prefix libomp)/lib/libomp.dylib \
  ..

# For High Sierra or earlier (<= 10.13)
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
```

(continues on next page)

(continued from previous page)

```
cmake -DUSE_MPI=ON ..
make -j4
```

1.7 Build GPU Version

1.7.1 Linux

On Linux GPU version of LightGBM 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
cmake -DUSE_GPU=1 ..
# if you have installed NVIDIA CUDA to a customized location, you should specify␣
↪paths to OpenCL headers and library like the following:
# cmake -DUSE_GPU=1 -DOpenCL_LIBRARY=/usr/local/cuda/lib64/libOpenCL.so -DOpenCL_
↪INCLUDE_DIR=/usr/local/cuda/include/ ..
make -j4
```

1.7.2 Windows

On Windows GPU version of LightGBM 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`.

4. Run the following commands:

```
Set BOOST_ROOT=C:\local\boost_1_63_0\
Set BOOST_LIBRARYDIR=C:\local\boost_1_63_0\lib64-msvc-14.0
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -DCMAKE_GENERATOR_PLATFORM=x64 -DUSE_GPU=1 ..
cmake --build . --target ALL_BUILD --config Release
```

   Note: `C:\local\boost_1_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). You also can set them to the environment variable to avoid `Set ...` commands when build.

1.7.3 Docker

Refer to GPU Docker folder.

1.8 Build HDFS Version

Note: Installation process of HDFS version is untested.

1.8.1 Linux

On Linux HDFS version of LightGBM can be built using **CMake** and **gcc** or **Clang**.

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
```

1.9 Build Java Wrapper

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

1.9.1 Windows

On Windows 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:

```
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build
cd build
cmake -DCMAKE_GENERATOR_PLATFORM=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/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:

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

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)** for its better multithreading efficiency in **Windows** for many-core systems (see FAQ Question 4 and Question 8).

Also, you may want to read gcc Tips.

1.9.2 Linux

On Linux 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:

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

1.9.3 macOS

On macOS 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 (3.12 or higher):

```
brew install cmake
```

2. Install **OpenMP**:

```
brew install libomp
```

3. Run the following commands:

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

# For Mojave (10.14)
cmake \
  -DUSE_SWIG=ON \
  -DAPPLE_OUTPUT_DYLIB=ON \
  -DOpenMP_C_FLAGS="-Xpreprocessor -fopenmp -I$(brew --prefix libomp)/include" \
  -DOpenMP_C_LIB_NAMES="omp" \
  -DOpenMP_CXX_FLAGS="-Xpreprocessor -fopenmp -I$(brew --prefix libomp)/include" \
  -DOpenMP_CXX_LIB_NAMES="omp" \
  -DOpenMP_omp_LIBRARY=$(brew --prefix libomp)/lib/libomp.dylib \
  ..

# For High Sierra or earlier (<= 10.13)
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.

CHAPTER 2

Quick Start

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 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
- Parallel Learning

CHAPTER 3

Python-package Introduction

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

List of other helpful links

- Python Examples
- Python API
- Parameters Tuning

3.1 Install

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

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

Refer to Python-package folder for the installation guide.

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

```python
import lightgbm as lgb
```

3.2 Data Interface

The LightGBM Python module can load data from:

- libsvm/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.

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

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

To load a numpy array into Dataset:

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

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

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

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

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

Create validation data:

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

or

```
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:

```
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 coding, and is much faster than one-hot coding (about 8x speed-up).

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

Weights can be set when needed:

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

or

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

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

Memory 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 either a list of pairs or a dictionary to set Parameters. For instance:

- Booster parameters:

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

- You can also specify multiple eval metrics:

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

3.4 Training

Training a model requires a parameter list and data set:

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

After training, the model can be saved:

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

The trained model can also be dumped to JSON format:

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

A saved model can be loaded:

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

3.5 CV

Training with 5-fold CV:

```
num_round = 10
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:

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

The model will train until the validation score stops improving. Validation 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 iteration.

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.

3.7 Prediction

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

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

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

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

CHAPTER 4

Features

This is a conceptual overview of how LightGBM works[1]. We assume familiarity with decision tree boosting algorithms 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[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[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(#data)`
  - Computing the histogram has time complexity `O(#data)`, but this involves only a fast sum-up operation. Once the histogram is constructed, a histogram-based algorithm has time complexity `O(#bins)`, and `#bins` is far smaller than `#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 `#data` than its neighbor). It then can get histograms of its neighbor by histogram subtraction with small cost (`O(#bins)`)
- **Reduce memory usage**
  - Replaces continuous values with discrete bins. If `#bins` is small, can use small data type, e.g. uint8_t, to store training data
  - No need to store additional information for pre-sorting feature values
- **Reduce communication cost for parallel learning**

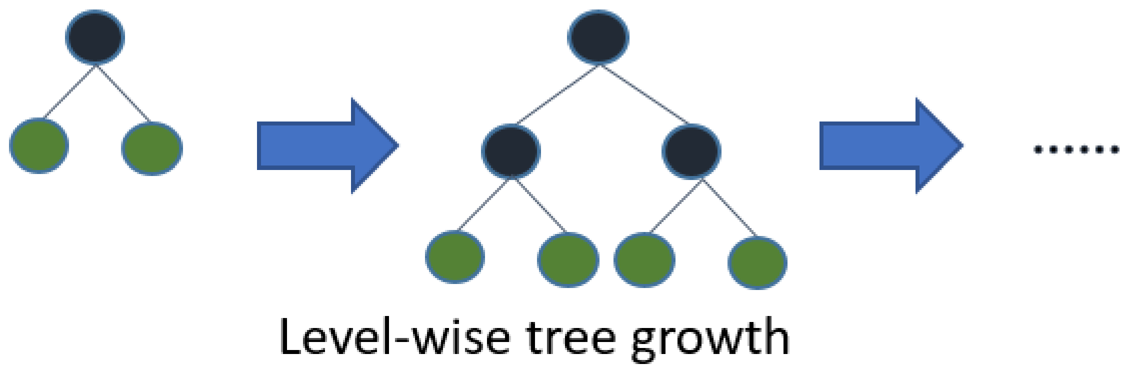
4.2 Sparse Optimization

- Need only `O(2 * #non_zero_data)` to construct histogram for sparse features

4.3 Optimization in Accuracy

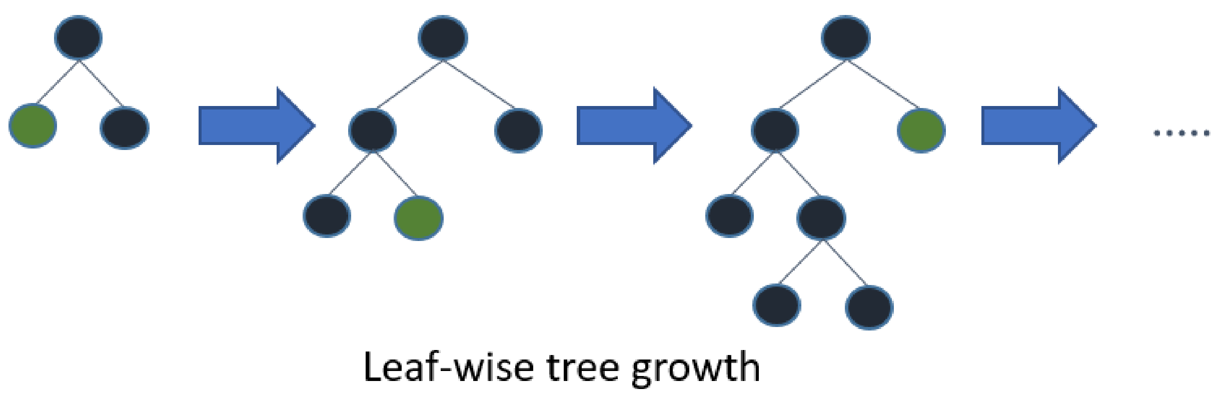
4.3.1 Leaf-wise (Best-first) Tree Growth

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



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.



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 * 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 (`sum_gradient / sum_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 parallel learning of LightGBM. LightGBM implement state-of-art algorithms*[9]*. These collective communication algorithms can provide much better performance than point-to-point communication.

4.5 Optimization in Parallel Learning

LightGBM provides the following parallel 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(#data)`. Thus, feature parallel cannot speed up well when `#data` is large.
- Need communication of split result, which costs about `O(#data / 8)` (one bit for one data).

Feature Parallel in LightGBM

Since feature parallel cannot speed up well when `#data` is large, we make a little change: 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 * #feature * #bin)`. If using collective communication algorithm (e.g. "All Reduce"), communication cost is about `O(2 * #feature * #bin)` (check cost of "All Reduce" in chapter 4.5 at *[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 use "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 * #feature * #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
- Fair
- Huber
- Poisson
- Quantile
- MAPE
- Kullback-Leibler
- Gamma
- Tweedie

For more details, please refer to Parameters.

4.8 Other Features

- Limit `max_depth` of tree while grows tree leaf-wise
- DART
- L1/L2 regularization
- Bagging
- Column (feature) sub-sample
- Continued train with input GBDT model
- Continued train with the input score file

- Weighted training
- Validation metric output during training
- Multi validation data
- Multi metrics
- Early stopping (both training and prediction)
- Prediction for leaf index

For more details, please refer to Parameters.

4.9 References

[1] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, Tie-Yan Liu. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." Advances in Neural Information Processing Systems 30 (NIPS 2017), pp. 3149-3157.

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

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

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

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

[6] Li, Ping, Qiang Wu, and Christopher J. Burges. "Mcrank: Learning to rank using multiple classification and gradient boosting." Advances in Neural Information Processing Systems 20 (NIPS 2007).

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

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

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

[10] Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, Tie-Yan Liu. "A Communication-Efficient Parallel Algorithm for Decision Tree." Advances in Neural Information Processing Systems 29 (NIPS 2016), pp. 1279-1287.

[11] Huan Zhang, Si Si and Cho-Jui Hsieh. "GPU Acceleration for Large-scale Tree Boosting." SysML Conference, 2018.

CHAPTER 5

Experiments

5.1 Comparison Experiment

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

5.1.1 Data

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

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

5.1.2 Environment

We ran all experiments on a single Linux server with the following specifications:

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

5.1.3 Baseline

We used xgboost as a baseline.

Both xgboost and LightGBM were built with OpenMP support.

5.1.4 Settings

We set up total 3 settings for experiments. The parameters of these settings are:

1. xgboost:

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

2. xgboost_hist (using histogram based algorithm):

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

3. LightGBM:

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

xgboost grows 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.

5.1.5 Result

Speed

We compared speed using only the training task without any test or metric output. We didn't count the time for IO.

The following table is the comparison of time cost:

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

LightGBM ran faster than xgboost on all experiment data sets.

Accuracy

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

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

Memory Consumption

We 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.

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

5.2 Parallel Experiment

5.2.1 Data

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

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

This data contains 13 integer features and 26 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 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.

5.2.2 Environment

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

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

5.2.3 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.

5.2.4 Results

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

The results show that LightGBM achieves a linear speedup with parallel learning.

5.3 GPU Experiments

Refer to GPU Performance.

CHAPTER 6

Parameters

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 *IO Parameters*
  - `refit`, for refitting existing models with new data, aliases: `refit_tree`
  - **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`, `gammma`, `tweedie`, `binary`, `multiclass`, `multiclassova`, `xentropy`, `xentlambda`, `lambdarank`, aliases: `objective_type`, `app`, `application`
  - regression application
    * `regression_l2`, L2 loss, aliases: `regression`, `mean_squared_error`, `mse`, `l2_root`, `root_mean_squared_error`, `rmse`
    * `regression_l1`, L1 loss, aliases: `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`, 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
    * `xentropy`, objective function for cross-entropy (with optional linear weights), aliases: `cross_entropy`
    * `xentlambda`, alternative parameterization of cross-entropy, aliases: `cross_entropy_lambda`
    * label is anything in interval [0, 1]
  - `lambdarank`, lambdarank application
    * label should be `int` type in lambdarank tasks, and larger number represents the higher relevance (e.g. 0:bad, 1:fair, 2:good, 3:perfect)
    * *label_gain* can be used to set the gain (weight) of `int` label
    * all values in `label` must be smaller than number of elements in `label_gain`

- `boosting` , default = `gbdt`, type = enum, options: `gbdt`, `gbrt`, `rf`, `random_forest`, `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

- `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 >= 0`
  - number of boosting iterations
  - **Note**: internally, LightGBM constructs `num_class * 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: `num_leaves > 1`
  - 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 Parallel 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 parallel learning, do not use all CPU cores because this will cause poor performance for the network communication
- `device_type` , default = `cpu`, type = enum, options: `cpu`, `gpu`, 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

6.3 Learning Control Parameters

- `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
  - `< 0` means no limit
- `min_data_in_leaf` , default = `20`, type = int, aliases: `min_data_per_leaf`, `min_data`, `min_child_samples`, constraints: `min_data_in_leaf >= 0`
  - minimal number of data in one leaf. Can be used to deal with over-fitting
- `min_sum_hessian_in_leaf` , default = `1e-3`, type = double, aliases: `min_sum_hessian_per_leaf`, `min_sum_hessian`, `min_hessian`, `min_child_weight`, constraints: `min_sum_hessian_in_leaf >= 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 < bagging_fraction <= 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
- `bagging_freq` , default = `0`, type = int, aliases: `subsample_freq`
  - frequency for bagging
  - `0` means disable bagging; `k` means perform bagging at every `k` iteration

- **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 part of features on each iteration if `feature_fraction` 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_seed` , default = `2`, type = int
  - random seed for `feature_fraction`
- `early_stopping_round` , default = `0`, type = int, aliases: `early_stopping_rounds`, `early_stopping`
  - will stop training if one metric of one validation data doesn't improve in last `early_stopping_round` rounds
  - `<= 0` means disable
- `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
- `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
- `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`
  - used only in `goss`
  - the retain ratio of large gradient data
- `other_rate` , default = `0.1`, type = double, constraints: `0.0 <= other_rate <= 1.0`
  - used only in `goss`
  - the retain ratio of small 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 the max threshold points in categorical features
- `cat_l2` , default = `10.0`, type = double, constraints: `cat_l2 >= 0.0`
  - used for the categorical features
  - L2 regularization in categorcial 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 in 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

- `feature_contri` , default = `None`, type = multi-double, aliases: `feature_contrib`, `fc`, `fp`, `feature_penalty`
  - used to control feature's split gain, will use `gain[i] = max(0, feature_contri[i]) * 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 <= refit_decay_rate <= 1.0`
  - decay rate of `refit` task, will use `leaf_output = refit_decay_rate * old_leaf_output + (1.0 - refit_decay_rate) * new_leaf_output` to refit trees
  - used only in `refit` task in CLI version or as argument in `refit` function in language-specific package

6.4 IO Parameters

- `verbosity` , default = `1`, type = int, aliases: `verbose`
  - controls the level of LightGBM's verbosity
  - `< 0`: Fatal, `= 0`: Error (Warning), `= 1`: Info, `> 1`: Debug
- `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`
- `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 histogram bins
  - setting this to larger value will give better training result, but will increase data loading time

  - set this to larger value if data is very sparse
- `histogram_pool_size` , default = `-1.0`, type = double, aliases: `hist_pool_size`
  - max cache size in MB for historical histogram
  - `< 0` means no limit
- `data_random_seed` , default = `1`, type = int, aliases: `data_seed`
  - random seed for data partition in parallel learning (excluding the `feature_parallel` mode)
- `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
- `snapshot_freq` , default = `-1`, type = int, aliases: `save_period`
  - 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
- `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_result` , default = `LightGBM_predict_result.txt`, type = string, aliases: `predict_result`, `prediction_result`, `predict_name`, `prediction_name`, `pred_name`, `name_pred`
  - filename of prediction result in `prediction` task
  - **Note**: can be used only in CLI version
- `initscore_filename` , default = `""`, type = string, aliases: `init_score_filename`, `init_score_file`, `init_score`, `input_init_score`
  - path of file with training initial scores
  - if `""`, will use `train_data_file` + `.init` (if exists)
  - **Note**: works only in case of loading data directly from file
- `valid_data_initscores` , default = `""`, type = string, aliases: `valid_data_init_scores`, `valid_init_score_file`, `valid_init_score`
  - path(s) of file(s) with validation initial scores
  - if `""`, will use `valid_data_file` + `.init` (if exists)
  - separate by `,` for multi-validation data
  - **Note**: works only in case of loading data directly from file
- `pre_partition` , default = `false`, type = bool, aliases: `is_pre_partition`
  - used for parallel learning (excluding the `feature_parallel` mode)

- `true` if training data are pre-partitioned, and different machines use different partitions

- `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
- `max_conflict_rate` , default = `0.0`, type = double, constraints: `0.0 <= max_conflict_rate < 1.0`
  - max conflict rate for bundles in EFB
  - set this to `0.0` to disallow the conflict and provide more accurate results
  - set this to a larger value to achieve faster speed
- `is_enable_sparse` , default = `true`, type = bool, aliases: `is_sparse`, `enable_sparse`, `sparse`
  - used to enable/disable sparse optimization
- `sparse_threshold` , default = `0.8`, type = double, constraints: `0.0 < sparse_threshold <= 1.0`
  - the threshold of zero elements percentage for treating a feature as a sparse one
- `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
- `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
- `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**: can be used only in CLI version; for language-specific packages you can use the correspondent function
- `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
  - **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
  - **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**

- `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 `#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
- `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
- `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
- `convert_model_language` , default = `""`, type = string
  - used only in `convert_model` task
  - only `cpp` is supported yet
  - 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

- `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` application
  - set this to `true` if training data are unbalanced
  - **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` application
  - weight of labels with positive class
  - **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` 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
  - used to control the variance of the tweedie distribution
  - set this closer to `2` to shift towards a **Gamma** distribution
  - set this closer to `1` to shift towards a **Poisson** distribution

- `max_position` , default = `20`, type = int, constraints: `max_position > 0`
  - used only in `lambdarank` application
  - optimizes NDCG at this position

- `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`
    * `l2_root`, root square loss, aliases: `root_mean_squared_error`, `rmse`
    * `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`
    * `map`, MAP, aliases: `mean_average_precision`
    * `auc`, AUC
    * `binary_logloss`, log loss, aliases: `binary`
    * `binary_error`, for one sample: `0` for correct classification, `1` for error classification

    * `multi_logloss`, log loss for multi-class classification, aliases: `multiclass`, `softmax`, `multiclassova`, `multiclass_ova`, `ova`, `ovr`
    * `multi_error`, error rate for multi-class classification
    * `xentropy`, cross-entropy (with optional linear weights), aliases: `cross_entropy`
    * `xentlambda`, "intensity-weighted" cross-entropy, aliases: `cross_entropy_lambda`
    * `kldiv`, Kullback-Leibler divergence, aliases: `kullback_leibler`
  - support multiple metrics, separated by `,`

- `metric_freq` , default = `1`, type = int, aliases: `output_freq`, constraints: `metric_freq > 0`
  - frequency for metric output
- `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 `,`

6.7 Network Parameters

- `num_machines` , default = `1`, type = int, aliases: `num_machine`, constraints: `num_machines > 0`
  - the number of machines for parallel learning application
  - this parameter is needed to be set in both **socket** and **mpi** versions
- `local_listen_port` , default = `12400`, 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 parallel learning application
  - each line contains one IP and one port for one machine. The format is `ip port` (space as a separator)
- `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)

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 in the same folder as the data file. In this case, LightGBM will auto load initial score file if it exists.

Otherwise, you should specify the path to the custom named file with initial scores by the `initscore_filename` *parameter*.

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 LambdaRank learning, it needs query information for training data. LightGBM uses an additional file to store query data, like the following:

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

It means first `27` lines samples belong to one query and next `18` lines belong to another, and so on.

Note: data should be ordered by the query.

If the name of data file is `train.txt`, the query file should be named as `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 `group_column` *parameter* in above.

CHAPTER 7

Parameters Tuning

This page contains parameters tuning guides for different scenarios.

List of other helpful links

- Parameters
- Python API

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 `num_leaves = 2^(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 `num_leaves`, we should let it be smaller than `2^(max_depth)`. For example, when the `max_depth=7` the depth-wise tree can get good accuracy, but setting `num_leaves` to `127` may cause over-fitting, and setting it to `70` or `80` may get better accuracy than depth-wise.
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 `num_leaves`. Setting it to a large value can avoid growing too deep a tree, but may cause under-fitting. In practice, setting it to hundreds or thousands is enough for a large dataset.
3. `max_depth`. You also can use `max_depth` to limit the tree depth explicitly.

7.2 For Faster Speed

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

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

CHAPTER 8

Python API

8.1 Data Structure API

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*)

Bases: `object`

Dataset in LightGBM.

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 size for Dataset.
- **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 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.
- **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.

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 size for Dataset.
- **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*

dump_text (*filename*)
Save Dataset to a text file.

This format cannot be loaded back in by LightGBM, but is useful for debugging purposes.

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

Returns **self** – Returns 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_penalty ()
Get the feature penalty of the Dataset.

Returns **feature_penalty** – Feature penalty for each feature in the Dataset.

Return type numpy array or None

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

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

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

Return type numpy array

get_group ()
Get the group of the Dataset.

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

Return type numpy array 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_monotone_constraints ()
Get the monotone constraints of the Dataset.

Returns **monotone_constraints** – Monotone constraints: -1, 0 or 1, for each feature in the Dataset.

Return type numpy array 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.

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 size of each group.

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*

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

Bases: object

Booster in LightGBM.

Initialize the Booster.

Parameters

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

add_valid(*data*, *name*)

Add validation data.

Parameters

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

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*)

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.

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, train_data, and return (eval_name, eval_result, is_higher_better) or list of such tuples. 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. 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, train_data, and return (eval_name, eval_result, is_higher_better) or list of such tuples. 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 feature 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 number 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*

get_leaf_output(*tree_id*, *leaf_id*)

Get the output of a leaf.

Parameters

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

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

Return type float

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.

  Note: 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.

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*)
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.

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*, *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.

- **num_iteration** (*int or None, optional (default=None)*) – Limit number of iterations in the prediction. If None, if the best iteration exists, it is used; otherwise, all iterations are used. If <= 0, all iterations 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 explanation for your model's predictions using SHAP values like SHAP interaction values, you can install shap package (https://github.com/slundberg/shap).

- **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.

Return type numpy array

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*)
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.

Returns **self** – Returns self.

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 parallel 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*

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.

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

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

Return type bool

8.2 Training API

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

Parameters

- **params** (*dict*) – Parameters for training.
- **train_set** (Dataset) – Data to be trained 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.
- **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. 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 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.
- **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. 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 dictionary used to store all evaluation results of all the items in `valid_sets`.

  #### Example

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

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

  #### Example

  With `verbose_eval` = 4 and at least one item in `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. You can still use _InnerPredictor as `init_model` for future continue training.
- **callbacks** (*list of callables or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

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

Return type *Booster*

`lightgbm.`**`cv`**(*params*, *train_set*, *num_boost_round=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*)

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 (http://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)*) – Custom objective function.
- **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. 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 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.

- **early_stopping_rounds** (*int or None, optional (default=None)*) – Activates 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. 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.

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

Return type dict

8.3 Scikit-learn API

class lightgbm.**LGBMModel**(*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: `object`

Implementation of the scikit-learn API for LightGBM.

Construct a gradient boosting model.

Parameters

- **boosting_type** (*string, optional (default='gbdt')*) – 'gbdt', traditional Gradient Boosting Decision Tree. 'dart', Dropouts meet Multiple Additive Regression Trees. 'goss', Gradient-based One-Side Sampling. 'rf', Random Forest.
- **num_leaves** (*int, optional (default=31)*) – Maximum tree leaves for base learners.
- **max_depth** (*int, optional (default=-1)*) – Maximum tree depth for base learners, -1 means no limit.
- **learning_rate** (*float, optional (default=0.1)*) – Boosting learning rate. 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. 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)*) – Frequence of subsample, <=0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int or None, optional (default=None)*) – Random number seed. If None, default seeds in C++ code will be 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.

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

`n_features_`
The number of features of fitted model.

Type int

`classes_`
The class label array (only for classification problem).

Type array of shape = [n_classes]

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

Type int

`best_score_`
The best score of fitted model.

Type dict or None

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

Type int or None

`objective_`
The concrete objective used while fitting this model.

Type string or callable

`booster_`
The underlying Booster of this model.

Type *Booster*

`evals_result_`
The evaluation results if `early_stopping_rounds` has been specified.

Type dict or None

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

Type array of shape = [n_features]

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

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

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

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

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

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

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

`best_iteration_`
Get the best iteration of fitted model.

`best_score_`
Get the best score of fitted model.

`booster_`
Get the underlying lightgbm Booster of this model.

`evals_result_`
Get the evaluation results.

`feature_importances_`
Get feature importances.

Note: Feature importance in sklearn interface used to normalize to 1, it's deprecated after 2.0.4 and is the same as Booster.feature_importance() now. `importance_type` attribute is passed to the function to configure the type of importance values to be extracted.

`fit`(*X*, *y*, *sample_weight=None*, *init_score=None*, *group=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*)
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 data 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_group`** (*`list of arrays or None, optional (default=None)`*) – Group data of eval data.
- **`eval_metric`** (*`string, list of strings, callable or None, optional (default=None)`*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. 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.
- **`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 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.
- **`callbacks`** (*`list of callback functions or None, optional (default=None)`*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **self** – Returns self.

Return type object

Note: Custom eval function expects a callable with following 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_bigger_better) or list of (eval_name, eval_result, is_bigger_better):

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

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

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

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

eval_name [string] The name of evaluation.

eval_result [float] The eval result.

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

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

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

n_features_

Get the number of features of fitted model.

objective_

Get the concrete objective used while fitting this model.

predict(*X*, *raw_score=False*, *num_iteration=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.
- **num_iteration** (*int or None, optional (default=None)*) – Limit number of iterations in the prediction. If None, if the best iteration exists, it is used; otherwise, all trees are used. If <= 0, all trees 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 explanation for your model's predictions using SHAP values like SHAP interaction values, you can install shap package (https://github.com/slundberg/shap).

- ****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 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]*) – If `pred_contrib=True`, the each 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

class lightgbm.**LGBMClassifier**(*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`, `object`

LightGBM classifier.

Construct a gradient boosting model.

Parameters

- **boosting_type** (*string, optional (default='gbdt')*) – 'gbdt', traditional Gradient Boosting Decision Tree. 'dart', Dropouts meet Multiple Additive Regression Trees. 'goss', Gradient-based One-Side Sampling. 'rf', Random Forest.
- **num_leaves** (*int, optional (default=31)*) – Maximum tree leaves for base learners.
- **max_depth** (*int, optional (default=-1)*) – Maximum tree depth for base learners, -1 means no limit.
- **learning_rate** (*float, optional (default=0.1)*) – Boosting learning rate. 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. 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)*) – Frequence of subsample, <=0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int or None, optional (default=None)*) – Random number seed. If None, default seeds in C++ code will be 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.

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

n_features_

The number of features of fitted model.

Type int

classes_

The class label array (only for classification problem).

Type array of shape = [n_classes]

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

Type int

best_score_
The best score of fitted model.

Type dict or None

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

Type int or None

objective_
The concrete objective used while fitting this model.

Type string or callable

booster_
The underlying Booster of this model.

Type *Booster*

evals_result_
The evaluation results if `early_stopping_rounds` has been specified.

Type dict or None

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

Type array of shape = [n_features]

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

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

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

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

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

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

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

---

best_iteration_
Get the best iteration of fitted model.

best_score_
Get the best score of fitted model.

booster_
Get the underlying lightgbm Booster of this model.

classes_
Get the class label array.

evals_result_
Get the evaluation results.

feature_importances_
Get feature importances.

Note: Feature importance in sklearn interface used to normalize to 1, it's deprecated after 2.0.4 and is the same as Booster.feature_importance() now. `importance_type` attribute is passed to the function to configure the type of importance values to be extracted.

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*)
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 data 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_group** (*list of arrays or None, optional (default=None)*) – Group data of eval data.
- **eval_metric** (*string, list of strings, callable or None, optional (default=None)*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. In either case, the `metric` from the model parameters will be evaluated and

used as well. Default: '12' 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.
- **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 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.
- **callbacks** (*list of callback functions or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **self** – Returns self.

Return type object

Note: Custom eval function expects a callable with following 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_bigger_better) or list of (eval_name, eval_result, is_bigger_better):

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

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

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

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

eval_name [string] The name of evaluation.

eval_result [float] The eval result.

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

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

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

n_classes_
Get the number of classes.

n_features_
Get the number of features of fitted model.

objective_
Get the concrete objective used while fitting this model.

predict (*X*, *raw_score=False*, *num_iteration=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.
- **num_iteration** (*int or None, optional (default=None)*) – Limit number of iterations in the prediction. If None, if the best iteration exists, it is used; otherwise, all trees are used. If <= 0, all trees 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 explanation for your model's predictions using SHAP values like SHAP interaction values, you can install shap package (https://github.com/slundberg/shap).

- ****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 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]*) – If `pred_contrib=True`, the each feature contributions for each sample.

predict_proba (*X*, *raw_score=False*, *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.
- **num_iteration** (*int or None, optional (default=None)*) – Limit number of iterations in the prediction. If None, if the best iteration exists, it is used; otherwise, all trees are used. If <= 0, all trees 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 explanation for your model's predictions using SHAP values like SHAP interaction values, you can install shap package (https://github.com/slundberg/shap).

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

Returns

- **predicted_probability** (*array-like of shape = [n_samples, n_classes]*) – The predicted probability for each class for each sample.
- **X_leaves** (*array-like of shape = [n_samples, n_trees * n_classes]*) – If `pred_leaf=True`, the predicted leaf every tree for each sample.
- **X_SHAP_values** (*array-like of shape = [n_samples, (n_features + 1) * n_classes]*) – If `pred_contrib=True`, the each 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

class lightgbm.**LGBMRegressor** (*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`, `object`

LightGBM regressor.

Construct a gradient boosting model.

Parameters

- **boosting_type** (*string, optional (default='gbdt')*) – 'gbdt', traditional Gradient Boosting Decision Tree. 'dart', Dropouts meet Multiple Additive Regression Trees. 'goss', Gradient-based One-Side Sampling. 'rf', Random Forest.
- **num_leaves** (*int, optional (default=31)*) – Maximum tree leaves for base learners.
- **max_depth** (*int, optional (default=-1)*) – Maximum tree depth for base learners, -1 means no limit.
- **learning_rate** (*float, optional (default=0.1)*) – Boosting learning rate. 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. 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)*) – Frequence of subsample, <=0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int or None, optional (default=None)*) – Random number seed. If None, default seeds in C++ code will be 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.

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

---

n_features_
The number of features of fitted model.

Type int

classes_
The class label array (only for classification problem).

Type array of shape = [n_classes]

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

Type int

best_score_
The best score of fitted model.

Type dict or None

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

Type int or None

objective_
The concrete objective used while fitting this model.

Type string or callable

booster_
The underlying Booster of this model.

Type *Booster*

evals_result_
The evaluation results if `early_stopping_rounds` has been specified.

Type dict or None

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

Type array of shape = [n_features]

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

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

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

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

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

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

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

`best_iteration_`
Get the best iteration of fitted model.

`best_score_`
Get the best score of fitted model.

`booster_`
Get the underlying lightgbm Booster of this model.

`evals_result_`
Get the evaluation results.

`feature_importances_`
Get feature importances.

Note: Feature importance in sklearn interface used to normalize to 1, it's deprecated after 2.0.4 and is the same as Booster.feature_importance() now. `importance_type` attribute is passed to the function to configure the type of importance values to be extracted.

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

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – Input feature matrix.
- **y** (*array-like of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Weights of training data.
- **init_score** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Init score of training data.

- **group** (*array-like or None, optional (default=None)*) – Group data of training data.
- **eval_set** (*list or None, optional (default=None)*) – A list of (X, y) tuple pairs to use as 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, list of strings, callable or None, optional (default=None)*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. In either case, the `metric` from the model parameters will be evaluated and used as well. Default: '12' 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.
- **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 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.
- **callbacks** (*list of callback functions or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **self** – Returns self.

Return type object

Note: Custom eval function expects a callable with following 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_bigger_better) or list of (eval_name, eval_result, is_bigger_better):

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

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

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

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

eval_name [string] The name of evaluation.

eval_result [float] The eval result.

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

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

`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

`n_features_`

Get the number of features of fitted model.

`objective_`

Get the concrete objective used while fitting this model.

`predict`(*X*, *raw_score=False*, *num_iteration=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.
- **`num_iteration`** (*`int or None, optional (default=None)`*) – Limit number of iterations in the prediction. If None, if the best iteration exists, it is used; otherwise, all trees are used. If <= 0, all trees 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 explanation for your model's predictions using SHAP values like SHAP interaction values, you can install shap package (https://github.com/slundberg/shap).

- `**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 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]*) – If `pred_contrib=True`, the each 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

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.

Construct a gradient boosting model.

Parameters

- **boosting_type** (*string, optional (default='gbdt')*) – 'gbdt', traditional Gradient Boosting Decision Tree. 'dart', Dropouts meet Multiple Additive Regression Trees. 'goss', Gradient-based One-Side Sampling. 'rf', Random Forest.
- **num_leaves** (*int, optional (default=31)*) – Maximum tree leaves for base learners.
- **max_depth** (*int, optional (default=-1)*) – Maximum tree depth for base learners, -1 means no limit.
- **learning_rate** (*float, optional (default=0.1)*) – Boosting learning rate. 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. 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)*) – Frequence of subsample, <=0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int or None, optional (default=None)*) – Random number seed. If None, default seeds in C++ code will be 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.

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

n_features_

The number of features of fitted model.

Type int

classes_
The class label array (only for classification problem).

Type array of shape = [n_classes]

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

Type int

best_score_
The best score of fitted model.

Type dict or None

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

Type int or None

objective_
The concrete objective used while fitting this model.

Type string or callable

booster_
The underlying Booster of this model.

Type *Booster*

evals_result_
The evaluation results if `early_stopping_rounds` has been specified.

Type dict or None

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

Type array of shape = [n_features]

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

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

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

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

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

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

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

---

best_iteration_
Get the best iteration of fitted model.

best_score_
Get the best score of fitted model.

booster_
Get the underlying lightgbm Booster of this model.

evals_result_
Get the evaluation results.

feature_importances_
Get feature importances.

Note: Feature importance in sklearn interface used to normalize to 1, it's deprecated after 2.0.4 and is the same as Booster.feature_importance() now. `importance_type` attribute is passed to the function to configure the type of importance values to be extracted.

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

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – Input feature matrix.
- **y** (*array-like of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Weights of training data.
- **init_score** (*array-like of shape = [n_samples] or None, optional (default=None)*) – Init score of training data.
- **group** (*array-like or None, optional (default=None)*) – Group data of training data.
- **eval_set** (*list or None, optional (default=None)*) – A list of (X, y) tuple pairs to use as 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, list of strings, callable or None, optional (default=None)*) – If string, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. 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** (*list of int, optional (default=[1])*) – 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.
- **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 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.
- **callbacks** (*list of callback functions or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.

Returns **self** – Returns self.

Return type object

Note: Custom eval function expects a callable with following 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_bigger_better) or list of (eval_name, eval_result, is_bigger_better):

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

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

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

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

eval_name [string] The name of evaluation.

eval_result [float] The eval result.

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

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

---

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

n_features_
Get the number of features of fitted model.

objective_
Get the concrete objective used while fitting this model.

predict (*X*, *raw_score=False*, *num_iteration=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.
- **num_iteration** (*int or None, optional (default=None)*) – Limit number of iterations in the prediction. If None, if the best iteration exists, it is used; otherwise, all trees are used. If <= 0, all trees 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 explanation for your model's predictions using SHAP values like SHAP interaction values, you can install shap package (https://github.com/slundberg/shap).

- ****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 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]*) – If `pred_contrib=True`, the each 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

8.4 Callbacks

lightgbm.**early_stopping**(*stopping_rounds*, *verbose=True*)
Create a callback that activates early stopping.

Note: 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.

Parameters

- **stopping_rounds** (*int*) – The possible number of rounds without the trend occurrence.
- **verbose** (*bool, optional (default=True)*) – Whether to print message with early stopping information.

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

Return type function

lightgbm.**print_evaluation**(*period=1*, *show_stdv=True*)
Create a callback that prints the evaluation results.

Parameters

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

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

Return type function

lightgbm.**record_evaluation**(*eval_result*)
Create a callback that records the evaluation history into `eval_result`.

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

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

Return type function

lightgbm.**reset_parameter**(***kwargs*)
Create a callback that resets the parameter after the first iteration.

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

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

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

Return type function

8.5 Plotting

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

Plot model's feature importances.

Parameters

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

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

Plot one metric during training.

Parameters

- **booster** (*dict or* LGBMModel) – Dictionary returned from `lightgbm.train()` or LGBMModel instance.
- **metric** (*string or None, optional (default=None)*) – The metric name to plot. Only one metric supported because different metrics have various scales. If None, first metric picked from dictionary (according to hashcode).
- **dataset_names** (*list of strings or None, optional (default=None)*) – List of the dataset names which are used to calculate metric to plot. If None, all datasets are used.
- **ax** (*matplotlib.axes.Axes or None, optional (default=None)*) – Target axes instance. If None, new figure and axes will be created.
- **xlim** (*tuple of 2 elements or None, optional (default=None)*) – Tuple passed to `ax.xlim()`.
- **ylim** (*tuple of 2 elements or None, optional (default=None)*) – Tuple passed to `ax.ylim()`.
- **title** (*string or None, optional (default="Metric during training")*) – Axes title. If None, title is disabled.
- **xlabel** (*string or None, optional (default="Iterations")*) – X-axis title label. If None, title is disabled.
- **ylabel** (*string or None, optional (default="auto")*) – Y-axis title label. If 'auto', metric name is used. If None, title is disabled.
- **figsize** (*tuple of 2 elements or None, optional (default=None)*) – Figure size.
- **grid** (*bool, optional (default=True)*) – Whether to add a grid for axes.

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

Return type matplotlib.axes.Axes

`lightgbm.plot_tree`(*booster*, *ax=None*, *tree_index=0*, *figsize=None*, *old_graph_attr=None*, *old_node_attr=None*, *old_edge_attr=None*, *show_info=None*, *precision=None*, ***kwargs*)

Plot specified tree.

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.
- **show_info** (*list of strings or None, optional (default=None)*) – What information should be shown in nodes. Possible values of list items: 'split_gain', 'internal_value', 'internal_count', 'leaf_count'.
- **precision** (*int or None, optional (default=None)*) – Used to restrict the display of floating point values to a certain precision.
- ****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 **ax** – The plot with single tree.

Return type matplotlib.axes.Axes

`lightgbm.`**`create_tree_digraph`**(*booster, tree_index=0, show_info=None, precision=None, old_name=None, old_comment=None, old_filename=None, old_directory=None, old_format=None, old_engine=None, old_encoding=None, old_graph_attr=None, old_node_attr=None, old_edge_attr=None, old_body=None, old_strict=False, **kwargs*)

Create a digraph representation of specified tree.

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

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. Possible values of list items: 'split_gain', 'internal_value', 'internal_count', 'leaf_count'.
- **precision** (*int or None, optional (default=None)*) – Used to restrict the display of floating point values to a certain precision.
- ****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

CHAPTER 9

Parallel Learning Guide

This is a guide for parallel learning of LightGBM.

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

9.1 Choose Appropriate Parallel Algorithm

LightGBM provides 3 parallel learning algorithms now.

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

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

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

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

9.2 Build Parallel Version

Default build version support parallel learning based on the socket.

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

9.3 Preparation

9.3.1 Socket Version

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

```
machine1_ip 12345
machine2_ip 12345
```

9.3.2 MPI Version

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

```
machine1_ip
machine2_ip
```

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

9.4 Run Parallel Learning

9.4.1 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`

9.4.2 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
   ```

9.4.3 Example

- A simple parallel example

CHAPTER 10

LightGBM GPU Tutorial

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

For Windows, please see GPU Windows Tutorial.

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

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

10.2 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):

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

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

```
git clone --recursive https://github.com/Microsoft/LightGBM
cd LightGBM
mkdir build ; cd build
cmake -DUSE_GPU=1 ..
# if you have installed NVIDIA CUDA to a customized location, you should specify␣
↪paths to OpenCL headers and library like the following:
# cmake -DUSE_GPU=1 -DOpenCL_LIBRARY=/usr/local/cuda/lib64/libOpenCL.so -DOpenCL_
↪INCLUDE_DIR=/usr/local/cuda/include/ ..
make -j$(nproc)
cd ..
```

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.

10.3 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):

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

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

You can read our Python-package Examples for more information on how to use the Python interface.

10.4 Dataset Preparation

Using the following commands to prepare the Higgs dataset:

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

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

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

GPU is enabled in the configuration file we just created by setting `device=gpu`. 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.

10.5 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
```

Also, you can compare the training speed with CPU:

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

10.6 Further Reading

- GPU Tuning Guide and Performance Comparison
- GPU SDK Correspondence and Device Targeting Table
- GPU Windows Tutorial

10.7 Reference

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

Huan Zhang, Si Si and Cho-Jui Hsieh. "GPU Acceleration for Large-scale Tree Boosting." SysML Conference, 2018.

CHAPTER 11

Advanced Topics

11.1 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.

11.2 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.

11.3 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 `max_position` to set the NDCG optimization position.

11.4 Parameters Tuning

- Refer to Parameters Tuning.

11.5 Parallel Learning

- Refer to Parallel Learning Guide.

11.6 GPU Support

- Refer to GPU Tutorial and GPU Targets.

11.7 Recommendations for gcc Users (MinGW, *nix)

- Refer to gcc Tips.

CHAPTER 12

LightGBM FAQ

Contents

12.1 Critical

Please post an issue in Microsoft/LightGBM repository for any LightGBM issues you encounter. For critical issues (crash, prediction error, nonsense outputs. . .), you may also ping a member of the core team according to the relevant area of expertise by mentioning them with the arobase (@) symbol:

- @guolinke **Guolin Ke** (C++ code / R-package / Python-package)
- @chivee **Qiwei Ye** (C++ code / Python-package)
- @Laurae2 **Damien Soukhavong** (R-package)
- @jameslamb **James Lamb** (R-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?)
- Is it specific to the compiler? (gcc or Clang version? MinGW or Visual Studio version?)
- Is it specific to your Operating System? (Windows? Linux? macOS?)
- Are you able to reproduce this issue with a simple case?
- Does the issue persist after removing all optimization flags and compiling LightGBM in debug mode?

When submitting issues, please keep in mind that this is largely a volunteer effort, and we may not be available 24/7 to provide support.

12.2 LightGBM

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

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

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

- **Question 4**: I am using Windows. Should I use Visual Studio or MinGW for compiling LightGBM?
- **Solution 4**: Visual Studio performs best for LightGBM.

- **Question 5**: When using LightGBM GPU, I cannot reproduce results over several runs.
- **Solution 5**: This is normal and expected behaviour, but you may try to use `gpu_use_dp = true` for reproducibility (see Microsoft/LightGBM#560). You may also use the CPU version.

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

- **Question 7**: I tried to use Random Forest mode, and LightGBM crashes!
- **Solution 7**: This is expected behaviour for arbitrary parameters. To enable Random Forest, you must use `bagging_fraction` and `feature_fraction` different from 1, along with a `bagging_freq`. This thread includes an example.

- **Question 8**: CPU usage is low (like 10%) in Windows when using LightGBM on very large datasets with many-core systems.
- **Solution 8**: Please use Visual Studio as it may be 10x faster than MinGW especially for very large trees.

- **Question 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.
```

- **Solution 9**: 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.

- **Question 10**: LightGBM crashes randomly with the error like this.

```
OMP: Error #15: Initializing libiomp5.dylib, but found libomp.dylib already␣
↪initialized.
OMP: Hint: This means that multiple copies of the OpenMP runtime have been linked␣
↪into the program. That is dangerous, since it can degrade performance or cause␣
↪incorrect results. The best thing to do is to ensure that only a single OpenMP␣
↪runtime is linked into the process, e.g. by avoiding static linking of the␣
↪OpenMP runtime in any library. As an unsafe, unsupported, undocumented␣
↪workaround you can set the environment variable KMP_DUPLICATE_LIB_OK=TRUE to␣
↪allow the program to continue to execute, but that may cause crashes or␣
↪silently produce incorrect results. For more information, please see http://www.
↪intel.com/software/products/support/.
```

- **Solution 10**: File extensions in the error message may differ depending on the operating system. This error means that you have multiple OpenMP libraries installed on your machine and they conflict with each other.

  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`.

  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
```

  If this is not your case, then you should find conflicting OpenMP library installations on your own and leave only one of them.

- **Question 11**: LightGBM hangs when multithreading (OpenMP) and using forking in Linux at the same time.
- **Solution 11**: Use `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 (ex: 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).

12.3 R-package

- **Question 1**: Any training command using LightGBM does not work after an error occurred during the training of a previous LightGBM model.
- **Solution 1**: Run `lgb.unloader(wipe = TRUE)` in the R console, and recreate the LightGBM datasets (this will wipe all LightGBM-related variables). Due to the pointers, choosing to not wipe variables will not fix the error. This is a known issue: Microsoft/LightGBM#698.

- **Question 2**: I used `setinfo`, tried to print my `lgb.Dataset`, and now the R console froze!
- **Solution 2**: Avoid printing the `lgb.Dataset` after using `setinfo`. This is a known bug: Microsoft/LightGBM#539.

12.4 Python-package

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

```
error: Error: setup script specifies an absolute path:
/Users/Microsoft/LightGBM/python-package/lightgbm/../../lib_lightgbm.so
setup() arguments must *always* be /-separated paths relative to the setup.py
→directory, *never* absolute paths.
```

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

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

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

but I'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 2**: Because LightGBM constructs bin mappers to build trees, and train and valid Datasets within one Booster share the same bin mappers, categorical features and feature names etc., the Dataset objects are constructed when 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.

- **Question 3**: I encounter segmentation faults (segfaults) randomly after installing LightGBM from PyPI using `pip install lightgbm`.
- **Solution 3**: 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.

CHAPTER 13

Development Guide

13.1 Algorithms

Refer to Features to understand important algorithms used in LightGBM.

13.2 Classes and Code Structure

13.2.1 Important Classes

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

13.2.2 Code Structure

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

13.2.3 Documents API

Refer to docs README.

13.3 C API

Refer to the comments in c_api.h.

13.4 High Level Language Package

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

13.5 Questions

Refer to FAQ.

Also feel free to open issues if you met problems.

CHAPTER 14

GPU Tuning Guide and Performance Comparison

14.1 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.

14.2 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.

14.3 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. For dataset with a mixture of sparse and dense features, you can control the `sparse_threshold` parameter to make sure there are enough dense features to process on the GPU. If you have categorical features, use the `categorical_column` option and input them into LightGBM directly; do not convert them into one-hot variables. Make sure to check the run log and look at the reported number of sparse and dense features.
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.

14.4 Performance Comparison

We evaluate the training performance of GPU acceleration on the following datasets:

Data	Task	Link	#Examples	#Features	Comments
Higgs	Binary classification	link1	10,500,000	28	use last 500,000 samples as test set
Epsilon	Binary classification	link2	400,000	2,000	use the provided test set
Bosch	Binary classification	link3	1,000,000	968	use the provided test set
Yahoo LTR	Learning to rank	link4	473,134	700	set1.train as train, set1.test as test
MS LTR	Learning to rank	link5	2,270,296	137	{S1,S2,S3} as train set, {S5} as test set
Expo	Binary classification (Categorical)	link6	11,000,000	700	use last 1,000,000 as test set

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.

Hardware	Peak FLOPS	Peak Memory BW	Cost (MSRP)
AMD Radeon RX 480	5,161 GFLOPS	256 GB/s	$199
NVIDIA GTX 1080	8,228 GFLOPS	320 GB/s	$499
2x Xeon E5-2683v3 (28 cores)	1,792 GFLOPS	133 GB/s	$3,692

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:

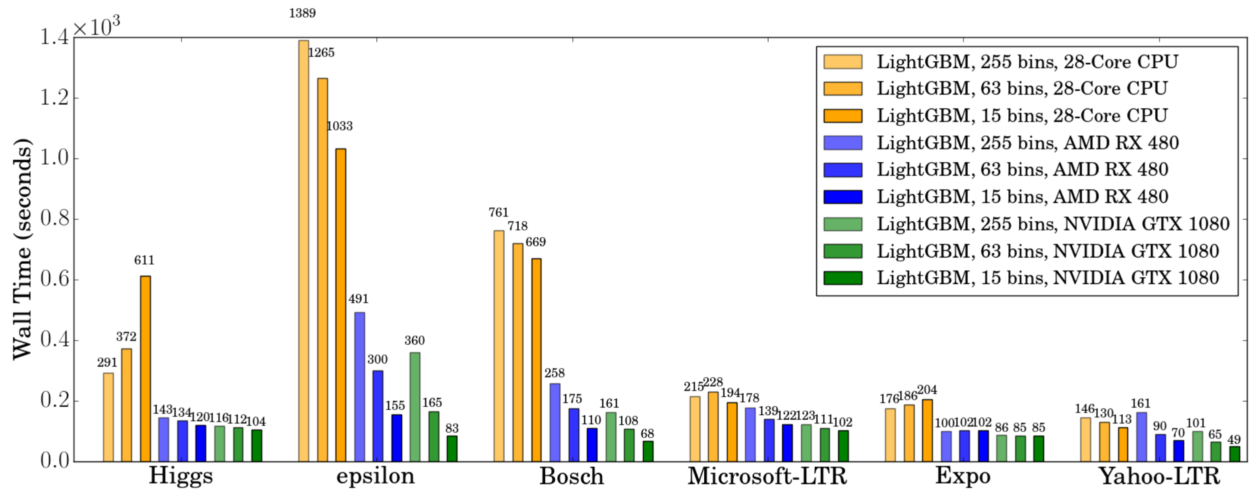
```
max_bin = 63
num_leaves = 255
num_iterations = 500
learning_rate = 0.1
tree_learner = serial
task = train
is_training_metric = false
min_data_in_leaf = 1
min_sum_hessian_in_leaf = 100
ndcg_eval_at = 1,3,5,10
sparse_threshold=1.0
device = gpu
gpu_platform_id = 0
gpu_device_id = 0
num_thread = 28
```

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 set `sparse_threshold=1`, and 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.

	CPU 255 bins	CPU 63 bins	CPU 15 bins	GPU 255 bins	GPU 63 bins	GPU 15 bins
Higgs AUC	0.845612	0.845239	0.841066	0.845612	0.845209	0.840748
Epsilon AUC	0.950243	0.949952	0.948365	0.950057	0.949876	0.948365
Yahoo-LTR $NDCG_1$	0.730824	0.730165	0.729647	0.730936	0.732257	0.73114
Yahoo-LTR $NDCG_3$	0.738687	0.737243	0.736445	0.73698	0.739474	0.735868
Yahoo-LTR $NDCG_5$	0.756609	0.755729	0.754607	0.756206	0.757007	0.754203
Yahoo-LTR $NDCG_{10}$	0.79655	0.795827	0.795273	0.795894	0.797302	0.795584
Expo AUC	0.776217	0.771566	0.743329	0.776285	0.77098	0.744078
MS-LTR $NDCG_1$	0.521265	0.521392	0.518653	0.521789	0.522163	0.516388
MS-LTR $NDCG_3$	0.503153	0.505753	0.501697	0.503886	0.504089	0.501691
MS-LTR $NDCG_5$	0.509236	0.510391	0.507193	0.509861	0.510095	0.50663
MS-LTR $NDCG_{10}$	0.527835	0.527304	0.524603	0.528009	0.527059	0.524722
Bosch AUC	0.718115	0.721791	0.716677	0.717184	0.724761	0.717005

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.

14.5 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.

Datasets	Higgs	Epsilon	Bosch	MS-LTR	Expo	Yahoo-LTR
GPU Memory Usage (MB)	611	901	1067	413	405	291

14.6 Further Reading

You can find more details about the GPU algorithm and benchmarks in the following article:

Huan Zhang, Si Si and Cho-Jui Hsieh. GPU Acceleration for Large-scale Tree Boosting. SysML Conference, 2018.

CHAPTER 15

GPU SDK Correspondence and Device Targeting Table

15.1 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:

SDK	CPU Intel/AMD	GPU Intel	GPU AMD	GPU NVIDIA
Intel SDK for OpenCL	Supported	Supported	Not Supported	Not Supported
AMD APP SDK *	Supported	Not Supported	Supported	Not Supported
NVIDIA CUDA Toolkit	Not Supported	Not Supported	Not Supported	Supported

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 for Linux from our GitHub repo.

15.2 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.

15.3 Examples

We provide test R code below, but you can use the language of your choice with the examples of your choices:

```
library(lightgbm)
data(agaricus.train, package = "lightgbm")
train <- agaricus.train
train$data[, 1] <- 1:6513
dtrain <- lgb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "lightgbm")
test <- agaricus.test
dtest <- lgb.Dataset.create.valid(dtrain, test$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):

```
> params <- list(objective = "regression",
+                metric = "rmse",
+                device = "gpu",
+                gpu_platform_id = 0,
+                gpu_device_id = 0,
+                nthread = 1,
+                boost_from_average = FALSE,
```

(continues on next page)

(continued from previous page)

```
+                 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 GPU Device: Oland, Vendor: Advanced Micro Devices, Inc.
[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 0.004211␣
↪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 max_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 max_depth=5
[2]:    test's rmse:0
```

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.

```
> 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
```

(continues on next page)

(continued from previous page)

```
[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 max_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 max_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.

CHAPTER 16

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).

16.1 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.

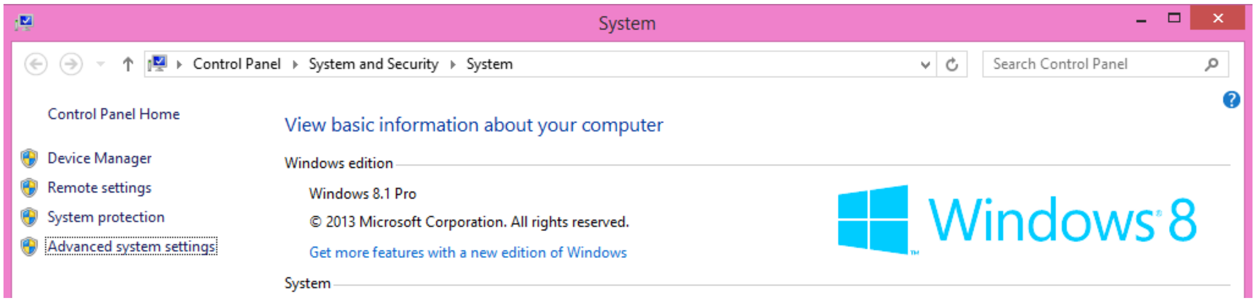
For this compilation tutorial, we are using AMD SDK for our OpenCL steps. However, you are free to use any OpenCL SDK you want, you just need to adjust the PATH correctly.

You will also need administrator rights. This will not work without them.

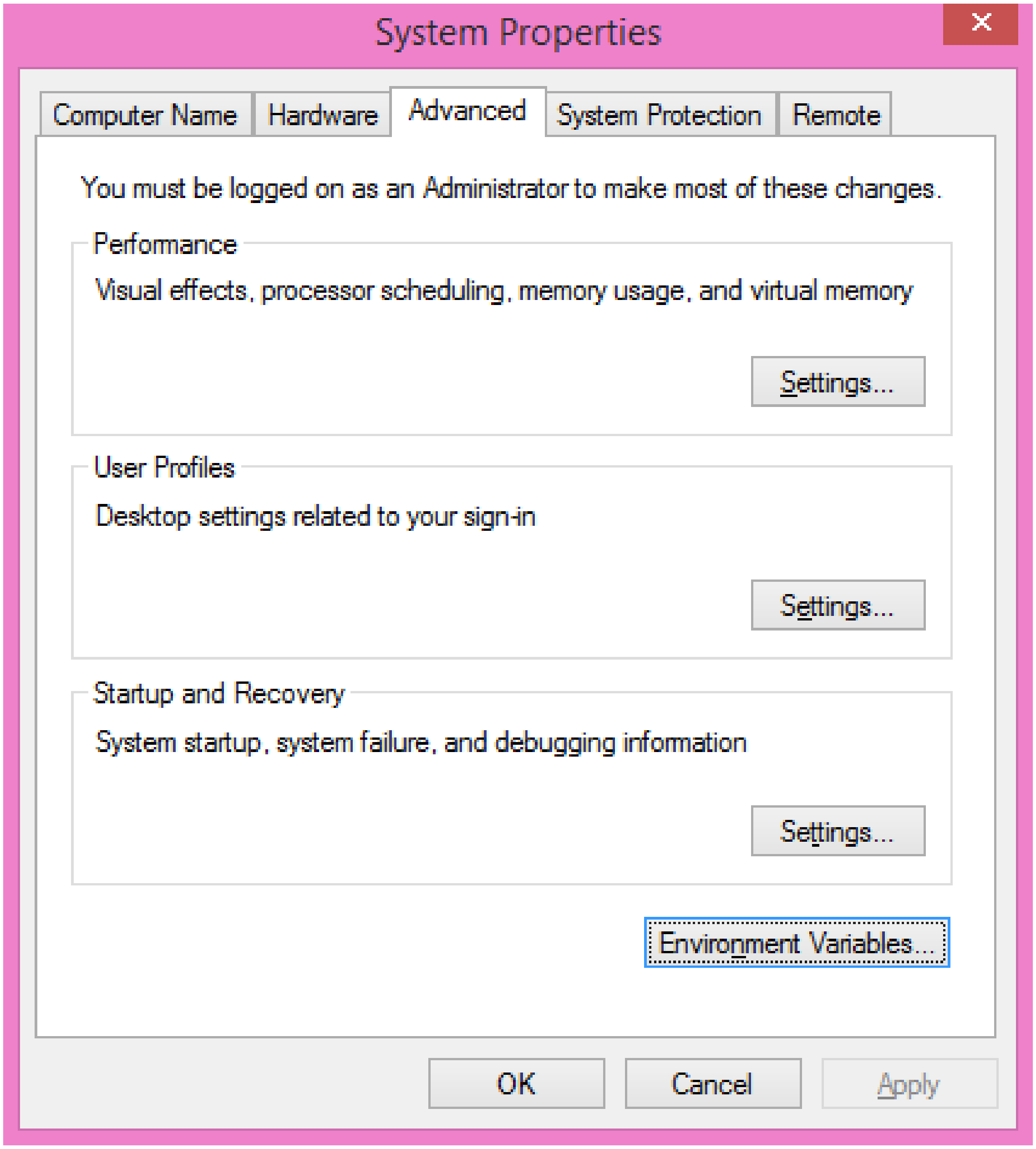
At the end, you can restore your original PATH.

16.1.1 Modifying PATH (for newbies)

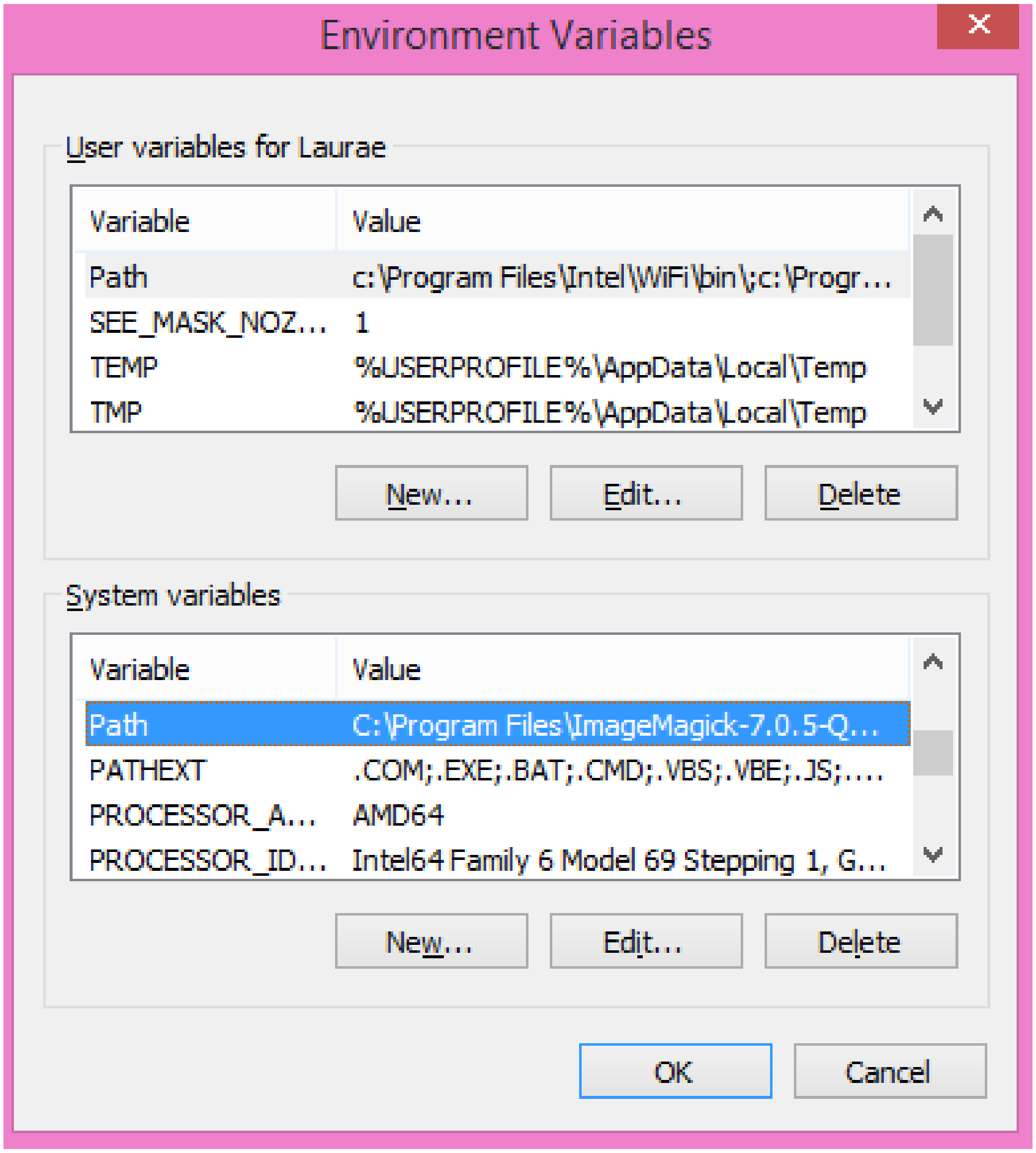
To modify PATH, just follow the pictures after going to the `Control Panel`:



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.

16.1.2 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 (you may want to replace the `OpenCL.dll` from 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.

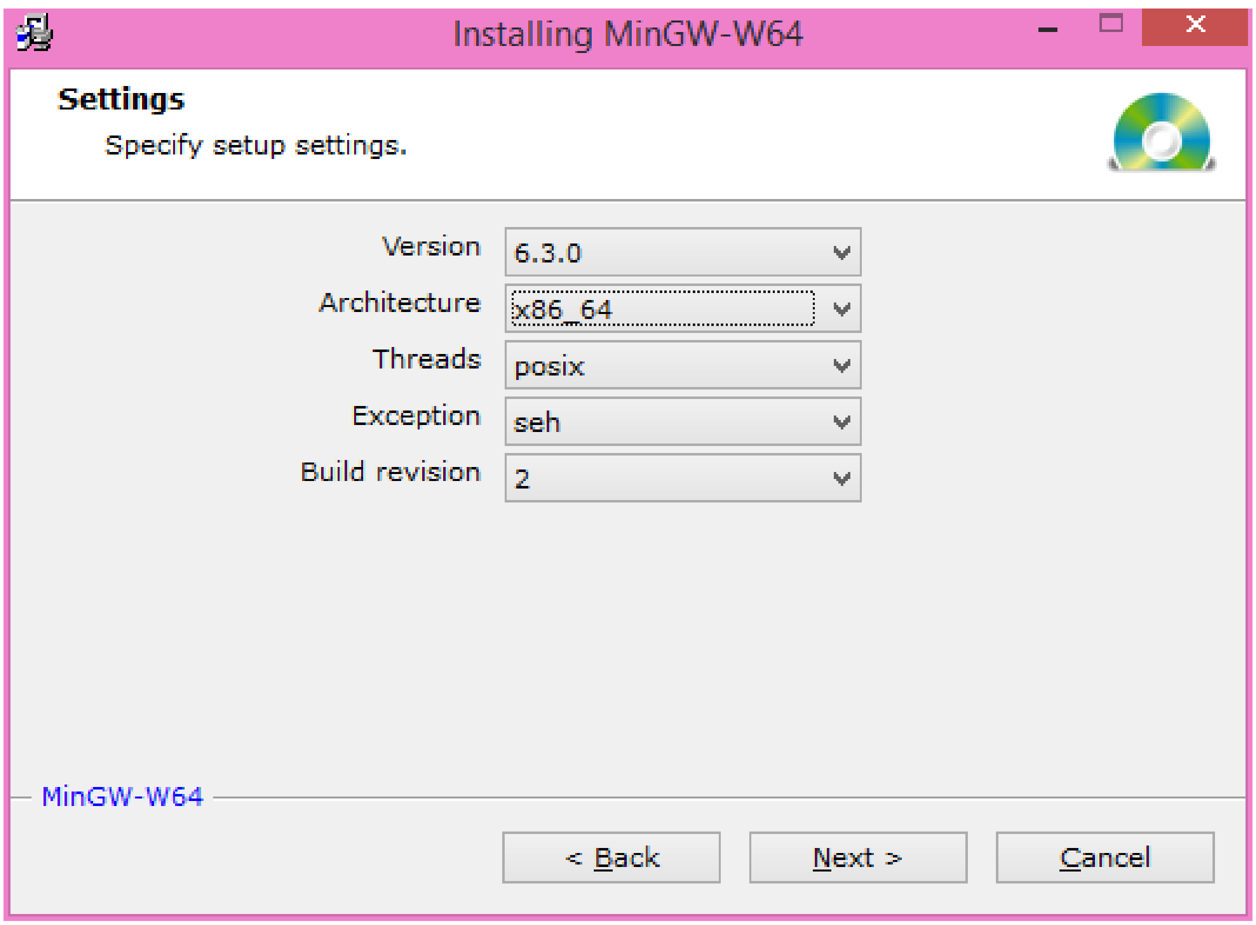
Further reading and correspondence table (especially if you intend to use cross-platform devices, like Intel CPU with AMD APP SDK): GPU SDK Correspondence and Device Targeting Table.

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).

16.1.3 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.



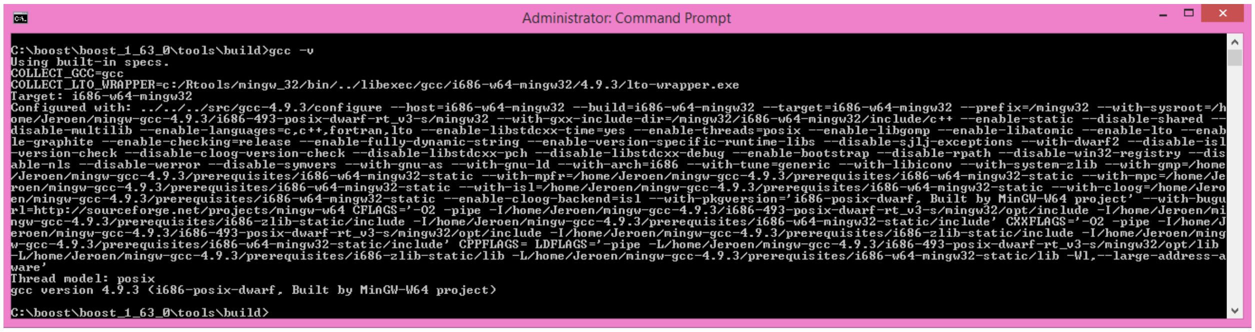
Then, add to your PATH the following (to adjust to your MinGW version):

```
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.

Quick installation of LightGBM can be done using:

```
devtools::install_github("Microsoft/LightGBM", subdir = "R-package")
```

16.1.4 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.

16.1.5 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

```
b2 install --build_dir="C:\boost\boost-build" --prefix="C:\boost\boost-build"␣
↪toolset=gcc --with=filesystem,system threading=multi --layout=system release
```

- 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
```

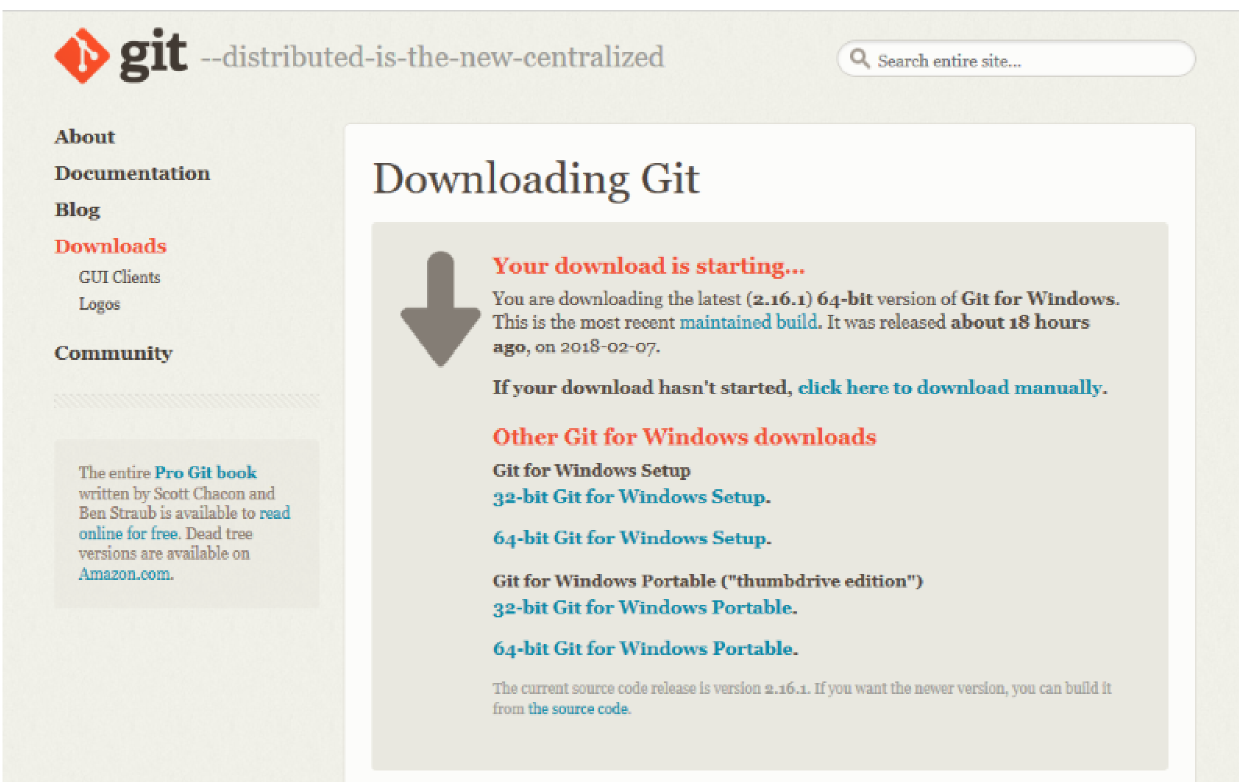
This is what you should (approximately) get at the end of Boost compilation:

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`

16.1.6 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
```

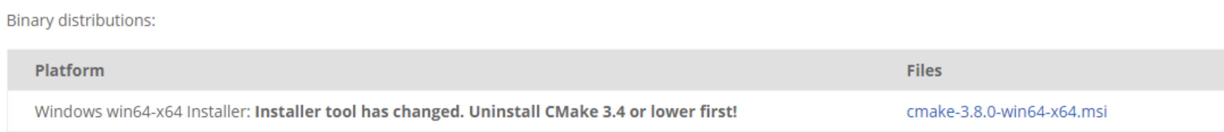
Your LightGBM repository copy should now be under `C:\github_repos\LightGBM`. You are free to use any folder you want, but you have to adapt.

Keep Git Bash open.

16.1.7 CMake Installation, Configuration, Generation

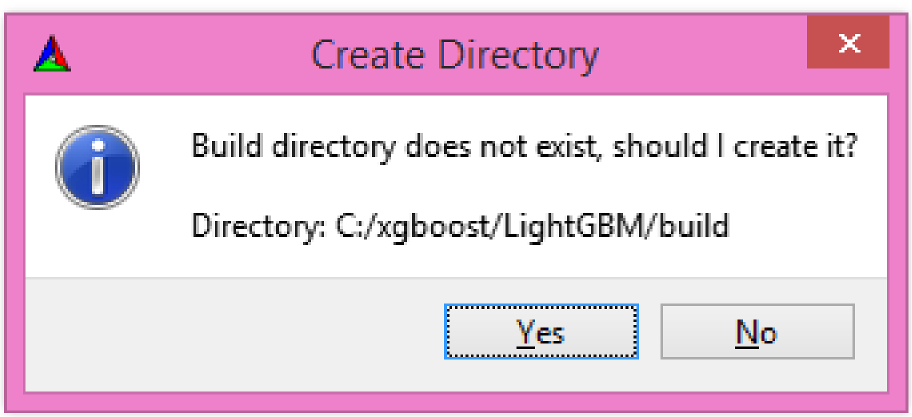
CLI / Python users only

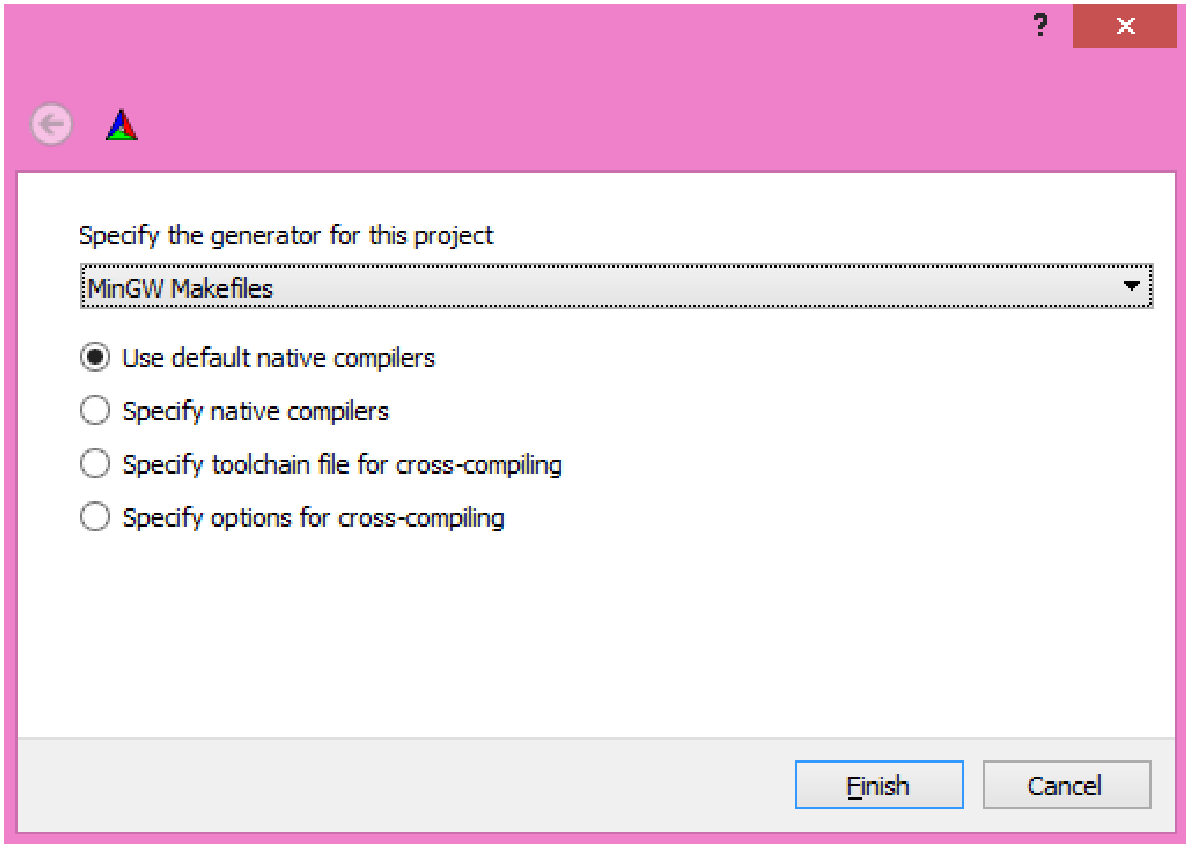
Installing CMake requires one download first and then a lot of configuration for LightGBM:


Binary distributions:

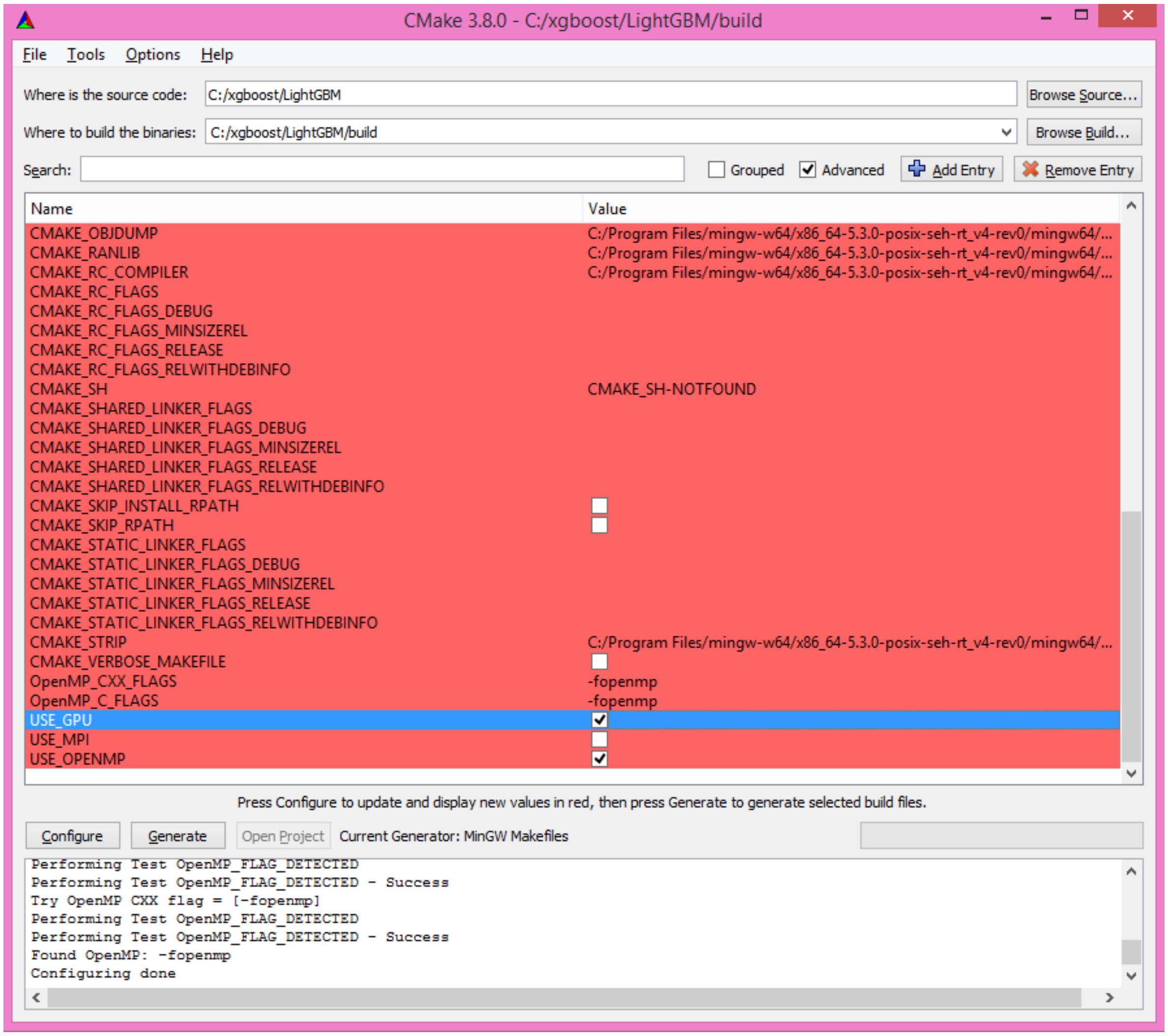
Platform	Files
Windows win64-x64 Installer: **Installer tool has changed. Uninstall CMake 3.4 or lower first!**	cmake-3.8.0-win64-x64.msi

- 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`



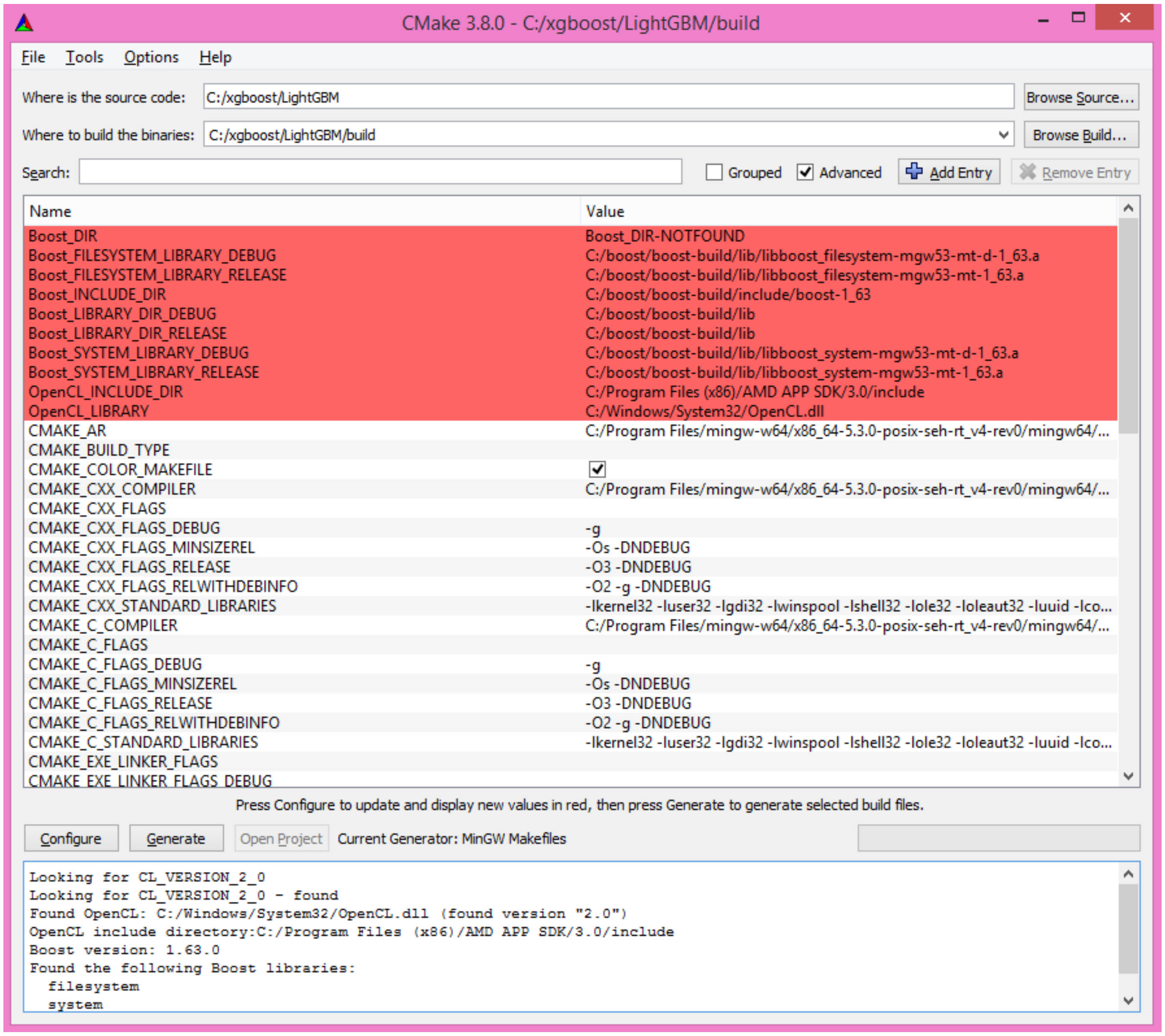


- Lookup for `USE_GPU` and check the checkbox



- 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.

16.1.8 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:

```
cd C:/github_repos/LightGBM/build
```

- If you did not close the Git Bash console previously, run this to get to the build folder:

```
cd LightGBM/build
```

- Setup MinGW as `make` using

```
alias make='mingw32-make'
```

  otherwise, beware error and name clash!

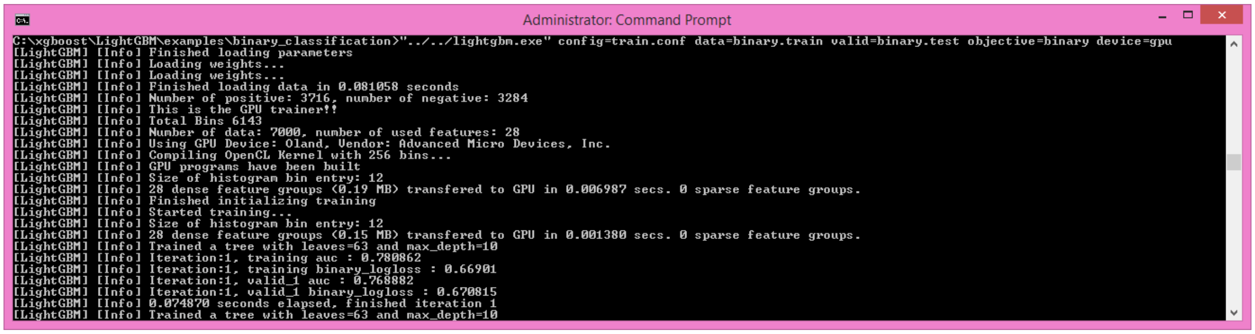
- In Git Bash, run `make` and see LightGBM being installing!

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
```

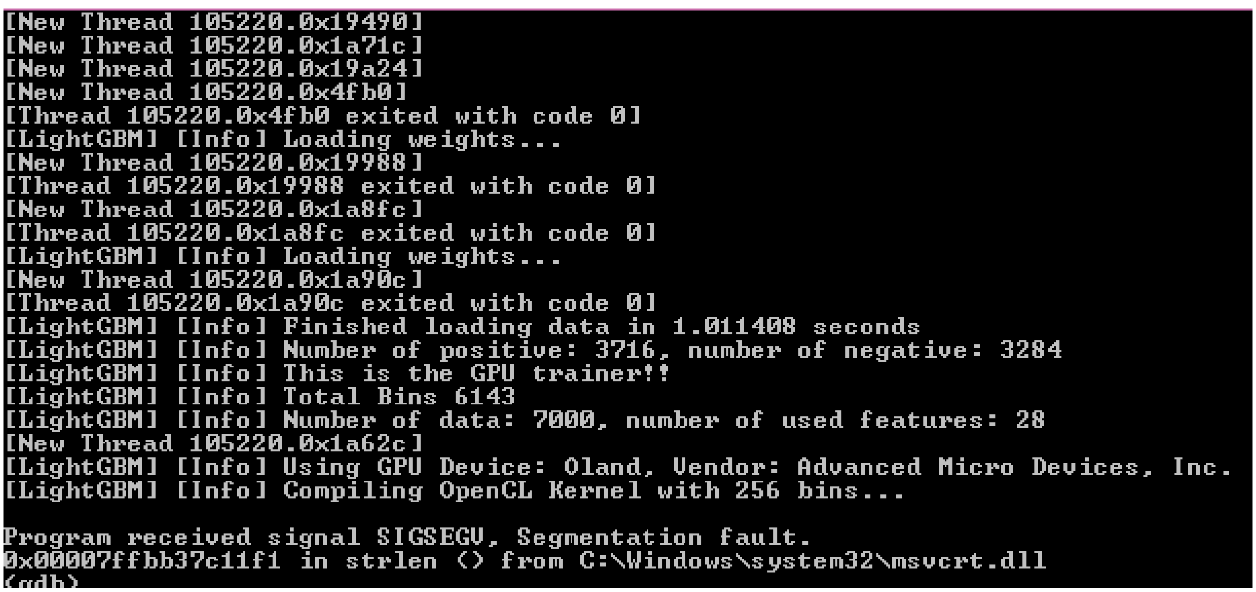


Congratulations for reaching this stage!

To learn how to target a correct CPU or GPU for training, please see: GPU SDK Correspondence and Device Targeting Table.

16.1.9 Debugging LightGBM Crashes in CLI

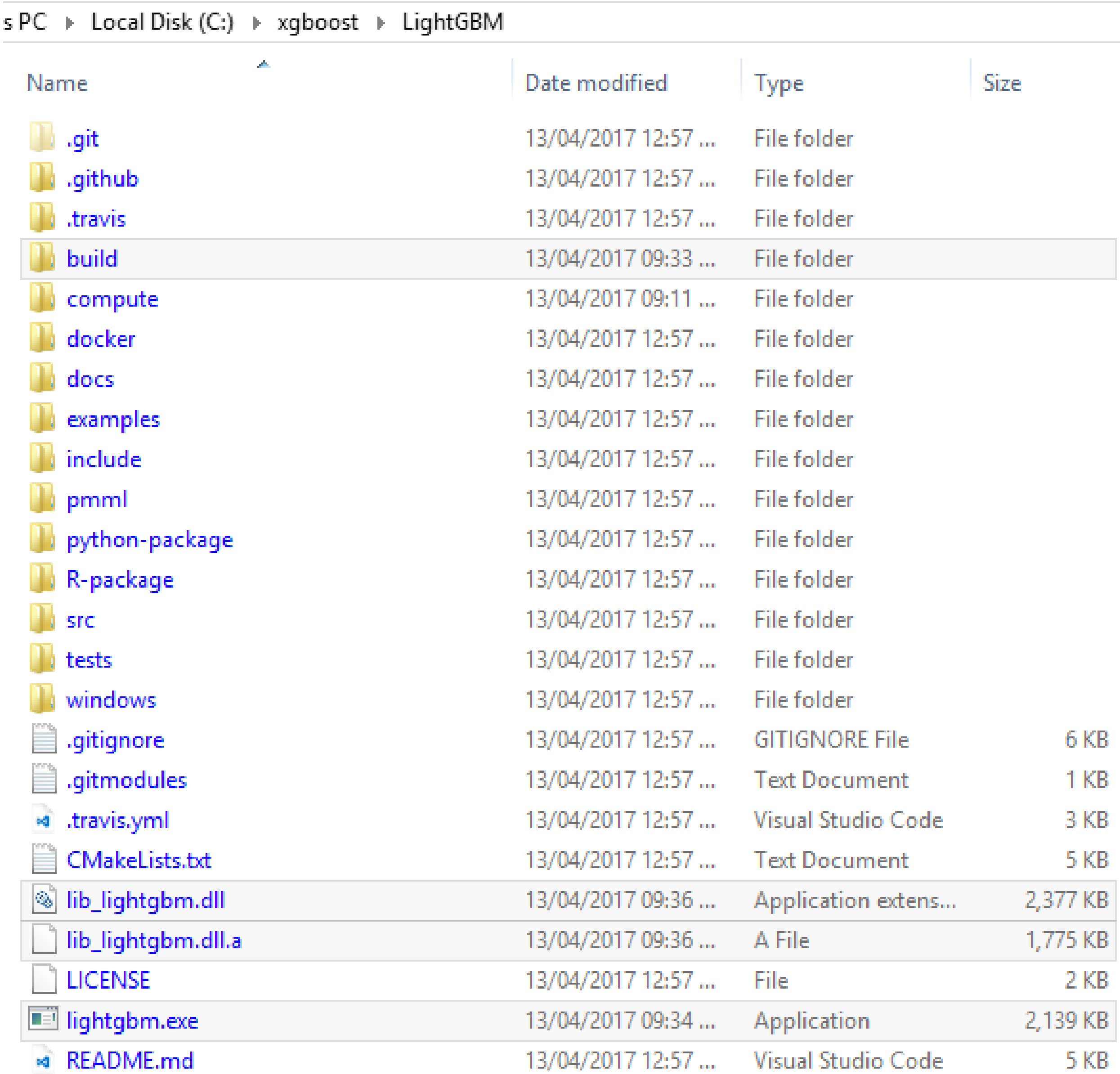
Now that you compiled LightGBM, you try it... and you always see a segmentation fault or an undocumented crash with GPU support:



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:

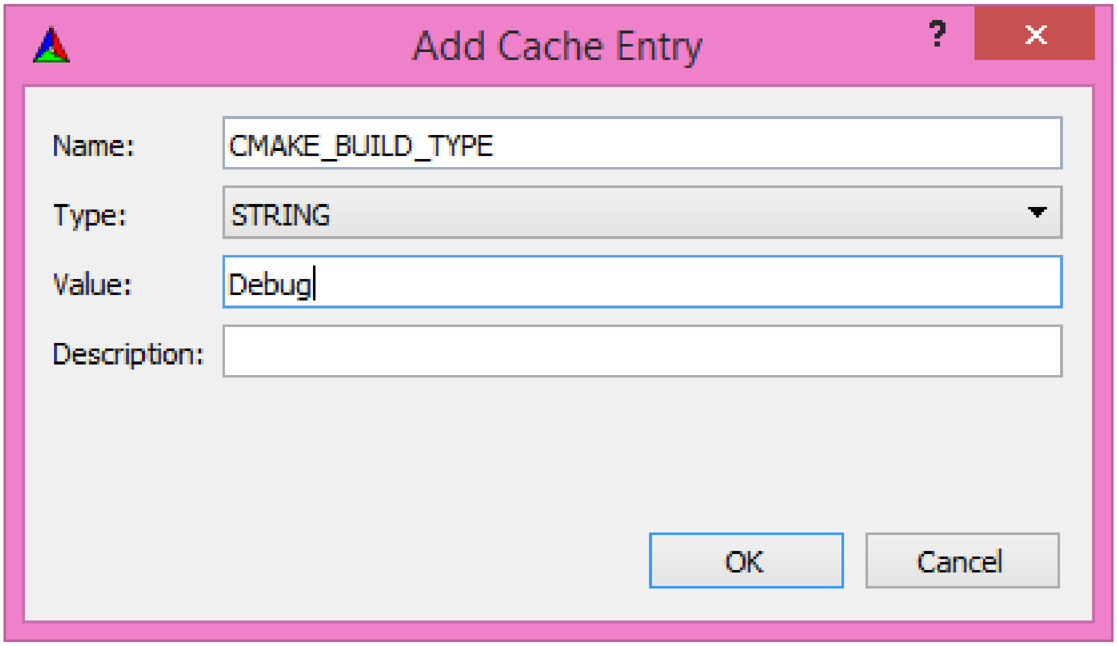
- Deleting `C:/github_repos/LightGBM/build` folder
- Deleting `lightgbm.exe`, `lib_lightgbm.dll`, and `lib_lightgbm.dll.a` files



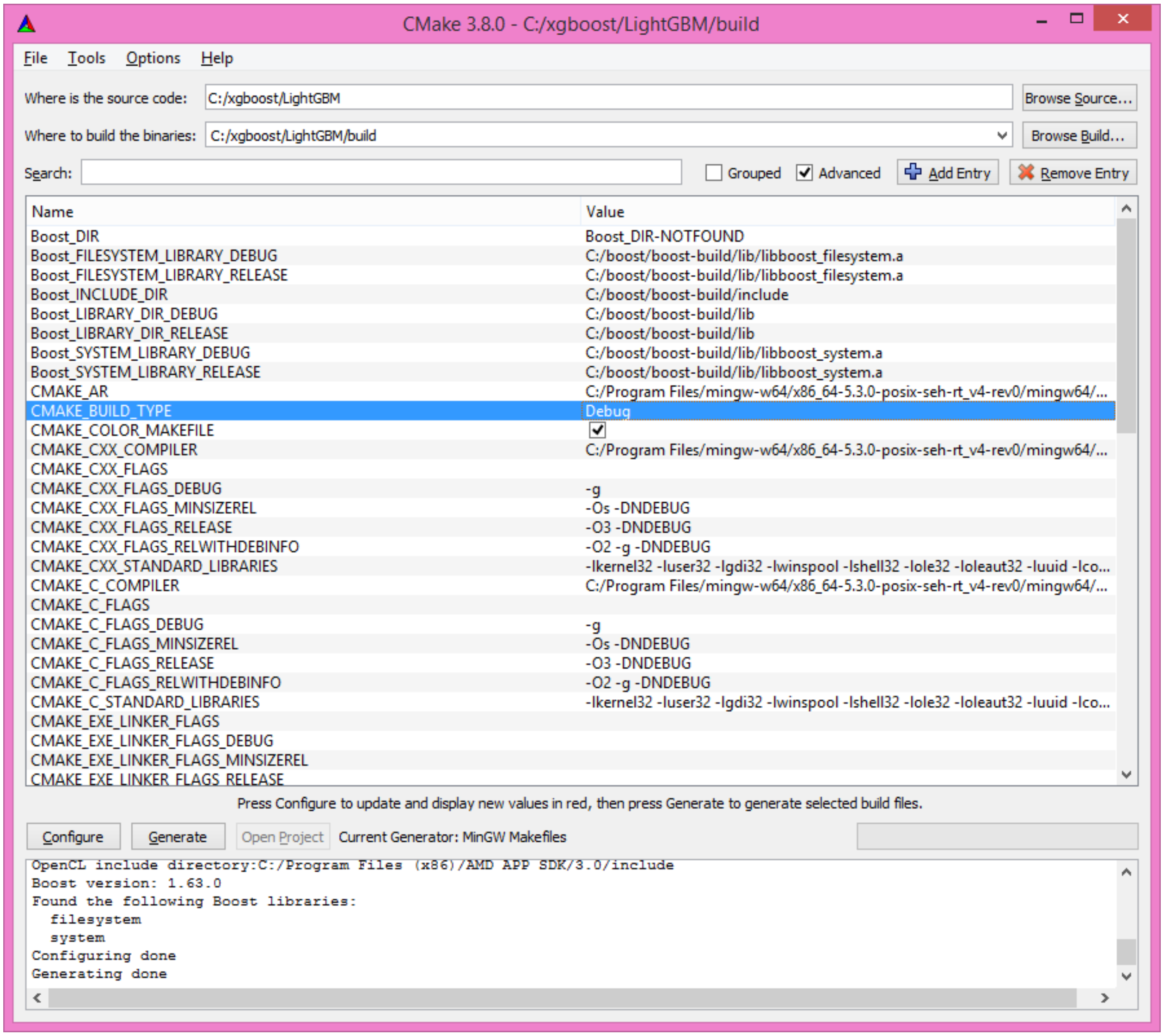
s PC ▸ Local Disk (C:) ▸ xgboost ▸ LightGBM

Name	Date modified	Type	Size
.git	13/04/2017 12:57 ...	File folder	
.github	13/04/2017 12:57 ...	File folder	
.travis	13/04/2017 12:57 ...	File folder	
build	13/04/2017 09:33 ...	File folder	
compute	13/04/2017 09:11 ...	File folder	
docker	13/04/2017 12:57 ...	File folder	
docs	13/04/2017 12:57 ...	File folder	
examples	13/04/2017 12:57 ...	File folder	
include	13/04/2017 12:57 ...	File folder	
pmml	13/04/2017 12:57 ...	File folder	
python-package	13/04/2017 12:57 ...	File folder	
R-package	13/04/2017 12:57 ...	File folder	
src	13/04/2017 12:57 ...	File folder	
tests	13/04/2017 12:57 ...	File folder	
windows	13/04/2017 12:57 ...	File folder	
.gitignore	13/04/2017 12:57 ...	GITIGNORE File	6 KB
.gitmodules	13/04/2017 12:57 ...	Text Document	1 KB
.travis.yml	13/04/2017 12:57 ...	Visual Studio Code	3 KB
CMakeLists.txt	13/04/2017 12:57 ...	Text Document	5 KB
lib_lightgbm.dll	13/04/2017 09:36 ...	Application extens...	2,377 KB
lib_lightgbm.dll.a	13/04/2017 09:36 ...	A File	1,775 KB
LICENSE	13/04/2017 12:57 ...	File	2 KB
lightgbm.exe	13/04/2017 09:34 ...	Application	2,139 KB
README.md	13/04/2017 12:57 ...	Visual Studio Code	5 KB

Once you removed the file, go into CMake, and follow the usual steps. Before clicking "Generate", click on "Add Entry":



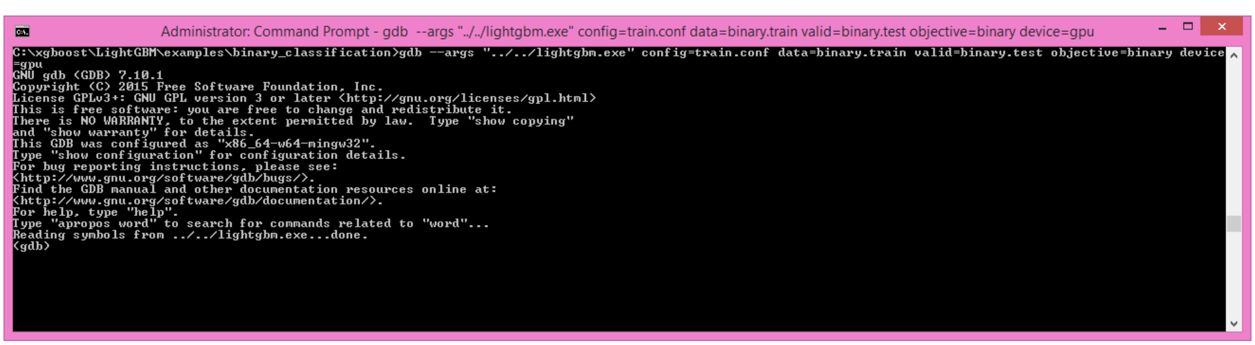
In addition, click on Configure and Generate:



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:

```
gdb --args "../../lightgbm.exe" config=train.conf data=binary.train valid=binary.test␣
↪objective=binary device=gpu
```



Type `run` and press the Enter key.

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] Using GPU Device: Oland, Vendor: Advanced Micro Devices, Inc.
[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  0x00007ffbb37c11f1 in strlen () from C:\Windows\system32\msvcrt.dll
#1  0x000000000048bbe5 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
#2  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
#3  boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/
↪include/boost/compute/detail/path.hpp:38
#4  0x000000000048eec3 in boost::compute::detail::program_binary_path (hash=
↪"d27987d5bd61e2d28cd32b8d7a7916126354dc81", create=create@entry=false)
    at C:/boost/boost-build/include/boost/compute/detail/path.hpp:46
#5  0x00000000004913de in boost::compute::program::load_program_binary (hash=
↪"d27987d5bd61e2d28cd32b8d7a7916126354dc81", ctx=...)
    at C:/boost/boost-build/include/boost/compute/program.hpp:605
#6  0x0000000000490ece in boost::compute::program::build_with_source (
    source="\n#ifndef _HISTOGRAM_256_KERNEL_\n#define _HISTOGRAM_256_KERNEL_\n\n
↪#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable\n#pragma OPENC
L EXTENSION cl_khr_global_int32_base_atomics : enable\n\n//"..., context=...,
    options=" -D POWER_FEATURE_WORKGROUPS=5 -D USE_CONSTANT_BUF=0 -D USE_DP_FLOAT=0 -
↪D CONST_HESSIAN=0 -cl-strict-aliasing -cl-mad-enable -cl-no-signed-zeros -c
l-fast-relaxed-math") at C:/boost/boost-build/include/boost/compute/program.hpp:549
#7  0x0000000000454339 in LightGBM::GPUTreeLearner::BuildGPUKernels () at␣
↪C:\LightGBM\src\treelearner\gpu_tree_learner.cpp:583
#8  0x00000000636044f2 in libgomp-1!GOMP_parallel () from C:\Program Files\mingw-
↪w64\x86_64-5.3.0-posix-seh-rt_v4-rev0\mingw64\bin\libgomp-1.dll
#9  0x0000000000455e7e in LightGBM::GPUTreeLearner::BuildGPUKernels␣
↪(this=this@entry=0x3b9cac0)
    at C:\LightGBM\src\treelearner\gpu_tree_learner.cpp:569
#10 0x0000000000457b49 in LightGBM::GPUTreeLearner::InitGPU (this=0x3b9cac0, platform_
↪id=<optimized out>, device_id=<optimized out>)
    at C:\LightGBM\src\treelearner\gpu_tree_learner.cpp:720
#11 0x0000000000410395 in LightGBM::GBDT::ResetTrainingData (this=0x1f26c90, config=
↪<optimized out>, train_data=0x1f28180, objective_function=0x1f280e0,
    training_metrics=std::vector of length 2, capacity 2 = {...}) at␣
↪C:\LightGBM\src\boosting\gbdt.cpp:98
#12 0x0000000000402e93 in LightGBM::Application::InitTrain (this=this@entry=0x23f9d0)␣
↪at C:\LightGBM\src\application\application.cpp:213
---Type <return> to continue, or q <return> to quit---
```

(continues on next page)

(continued from previous page)

```
#13 0x00000000004f0b55 in LightGBM::Application::Run (this=0x23f9d0) at C:/LightGBM/
↪include/LightGBM/application.h:84
#14 main (argc=6, argv=0x1f21e90) at C:\LightGBM\src\main.cpp:7
```

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"␣
↪config=train.conf data=binary.train valid=binary.test objective=binary device=gpu
GNU gdb (GDB) 7.10.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ../../lightgbm.exe...done.
(gdb) run
Starting program: C:\LightGBM\lightgbm.exe "config=train.conf" "data=binary.train"
↪"valid=binary.test" "objective=binary" "device=gpu"
[New Thread 105220.0x199b8]
[New Thread 105220.0x783c]
[Thread 105220.0x783c exited with code 0]
[LightGBM] [Info] Finished loading parameters
[New Thread 105220.0x19490]
[New Thread 105220.0x1a71c]
[New Thread 105220.0x19a24]
[New Thread 105220.0x4fb0]
[Thread 105220.0x4fb0 exited with code 0]
[LightGBM] [Info] Loading weights...
[New Thread 105220.0x19988]
[Thread 105220.0x19988 exited with code 0]
[New Thread 105220.0x1a8fc]
[Thread 105220.0x1a8fc exited with code 0]
[LightGBM] [Info] Loading weights...
[New Thread 105220.0x1a90c]
[Thread 105220.0x1a90c exited with code 0]
[LightGBM] [Info] Finished loading data in 1.011408 seconds
[LightGBM] [Info] Number of positive: 3716, number of negative: 3284
[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] Using GPU Device: Oland, Vendor: Advanced Micro Devices, Inc.
[LightGBM] [Info] Compiling OpenCL Kernel with 256 bins...

Program received signal SIGSEGV, Segmentation fault.
0x00007ffbb37c11f1 in strlen () from C:\Windows\system32\msvcrt.dll
(gdb) backtrace
```

(continues on next page)

(continued from previous page)

```
#0  0x00007ffbb37c11f1 in strlen () from C:\Windows\system32\msvcrt.dll
#1  0x000000000048bbe5 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
#2  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
#3  boost::compute::detail::appdata_path[abi:cxx11]() () at C:/boost/boost-build/
↪include/boost/compute/detail/path.hpp:38
#4  0x000000000048eec3 in boost::compute::detail::program_binary_path (hash=
↪"d27987d5bd61e2d28cd32b8d7a7916126354dc81", create=create@entry=false)
    at C:/boost/boost-build/include/boost/compute/detail/path.hpp:46
#5  0x00000000004913de in boost::compute::program::load_program_binary (hash=
↪"d27987d5bd61e2d28cd32b8d7a7916126354dc81", ctx=...)
    at C:/boost/boost-build/include/boost/compute/program.hpp:605
#6  0x0000000000490ece in boost::compute::program::build_with_source (
    source="\n#ifndef _HISTOGRAM_256_KERNEL_\n#define _HISTOGRAM_256_KERNEL_\n\n
↪#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable\n#pragma OPENCL␣
↪EXTENSION cl_khr_global_int32_base_atomics : enable\n\n//"..., context=...,
    options=" -D POWER_FEATURE_WORKGROUPS=5 -D USE_CONSTANT_BUF=0 -D USE_DP_FLOAT=0 -
↪D CONST_HESSIAN=0 -cl-strict-aliasing -cl-mad-enable -cl-no-signed-zeros -cl-fast-
↪relaxed-math") at C:/boost/boost-build/include/boost/compute/program.hpp:549
#7  0x0000000000454339 in LightGBM::GPUTreeLearner::BuildGPUKernels () at␣
↪C:\LightGBM\src\treelearner\gpu_tree_learner.cpp:583
#8  0x00000000636044f2 in libgomp-1!GOMP_parallel () from C:\Program Files\mingw-
↪w64\x86_64-5.3.0-posix-seh-rt_v4-rev0\mingw64\bin\libgomp-1.dll
#9  0x0000000000455e7e in LightGBM::GPUTreeLearner::BuildGPUKernels␣
↪(this=this@entry=0x3b9cac0)
    at C:\LightGBM\src\treelearner\gpu_tree_learner.cpp:569
#10 0x0000000000457b49 in LightGBM::GPUTreeLearner::InitGPU (this=0x3b9cac0, platform_
↪id=<optimized out>, device_id=<optimized out>)
    at C:\LightGBM\src\treelearner\gpu_tree_learner.cpp:720
#11 0x0000000000410395 in LightGBM::GBDT::ResetTrainingData (this=0x1f26c90, config=
↪<optimized out>, train_data=0x1f28180, objective_function=0x1f280e0,
    training_metrics=std::vector of length 2, capacity 2 = {...}) at␣
↪C:\LightGBM\src\boosting\gbdt.cpp:98
#12 0x0000000000402e93 in LightGBM::Application::InitTrain (this=this@entry=0x23f9d0)␣
↪at C:\LightGBM\src\application\application.cpp:213
---Type <return> to continue, or q <return> to quit---
#13 0x00000000004f0b55 in LightGBM::Application::Run (this=0x23f9d0) at C:/LightGBM/
↪include/LightGBM/application.h:84
#14 main (argc=6, argv=0x1f21e90) at C:\LightGBM\src\main.cpp:7
```

And open an issue in GitHub here with that log.

CHAPTER 17

Recommendations When Using gcc

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:

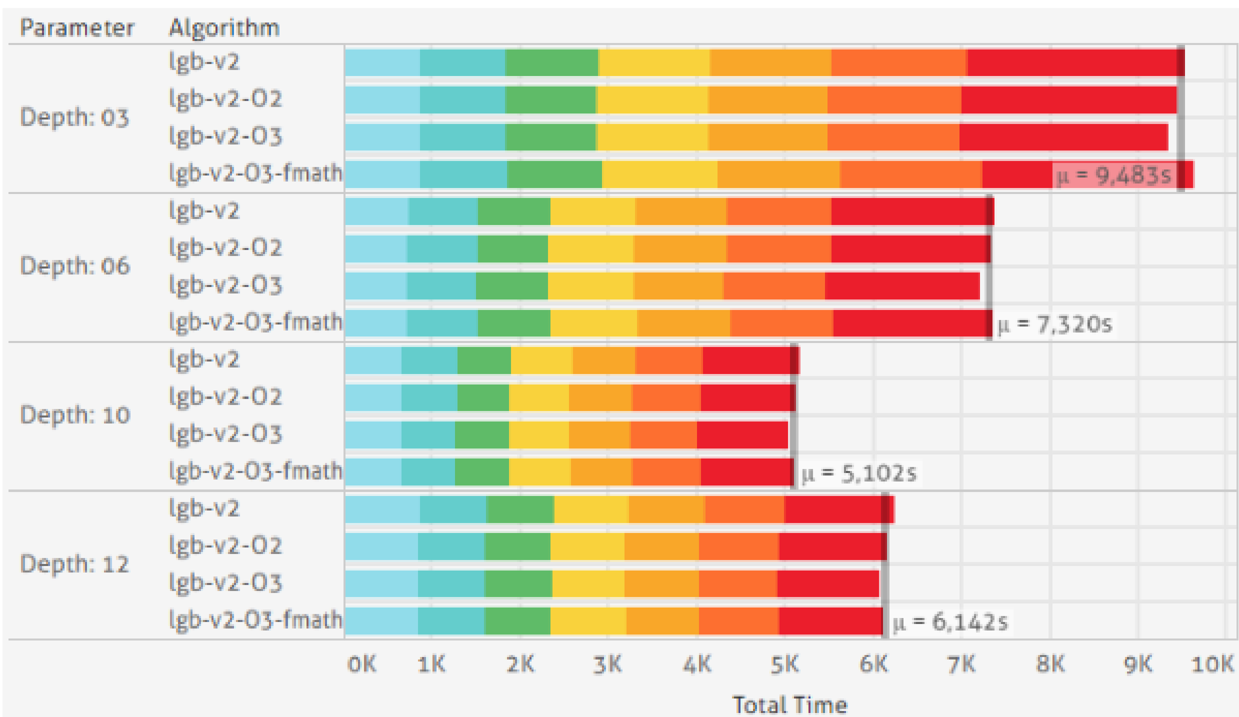
Compilation Flag	Performance Index
`-O2 -mtune=core2`	100.00%
`-O2 -mtune=native`	100.90%
`-O3 -mtune=native`	102.78%
`-O3 -ffast-math -mtune=native`	100.64%

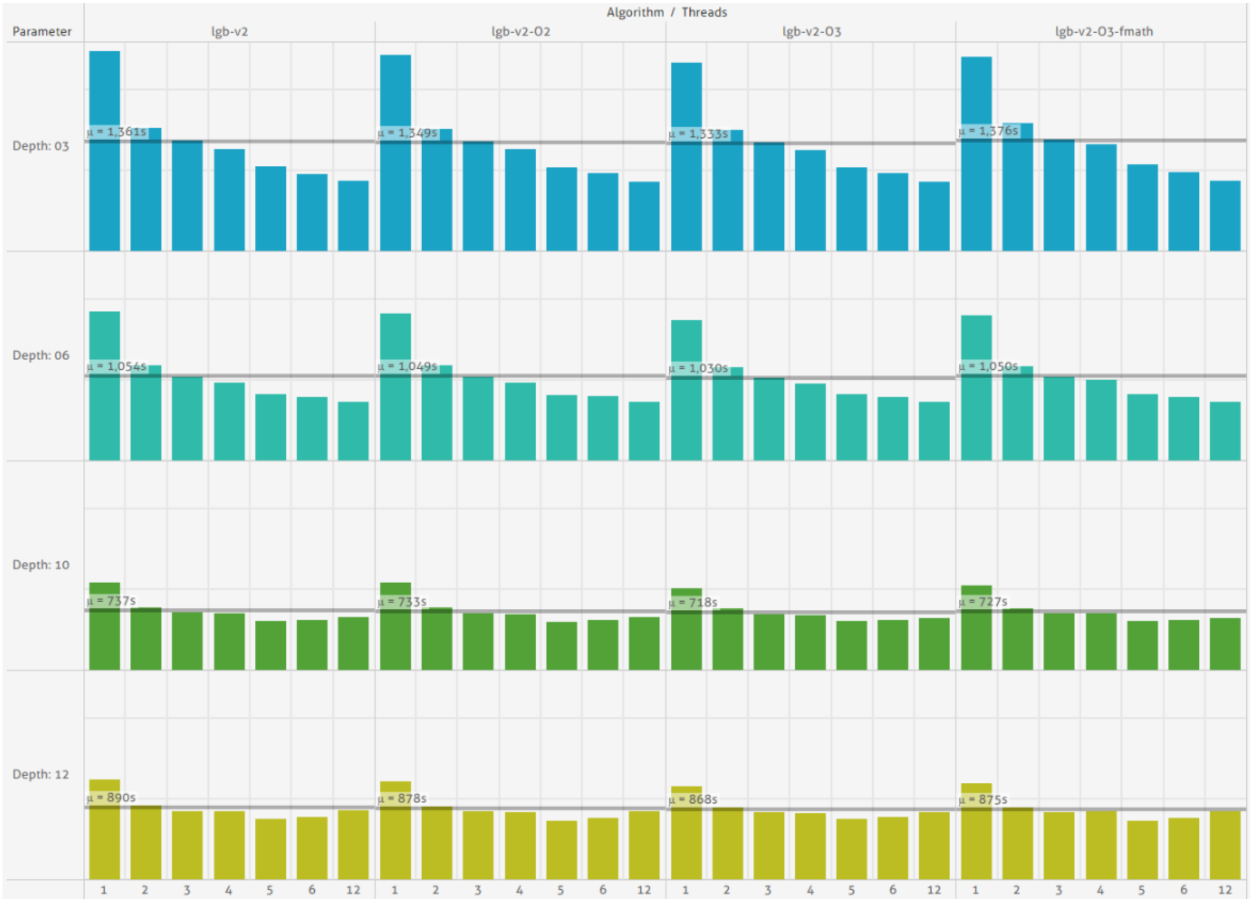
You can find more details on the experimentation below:

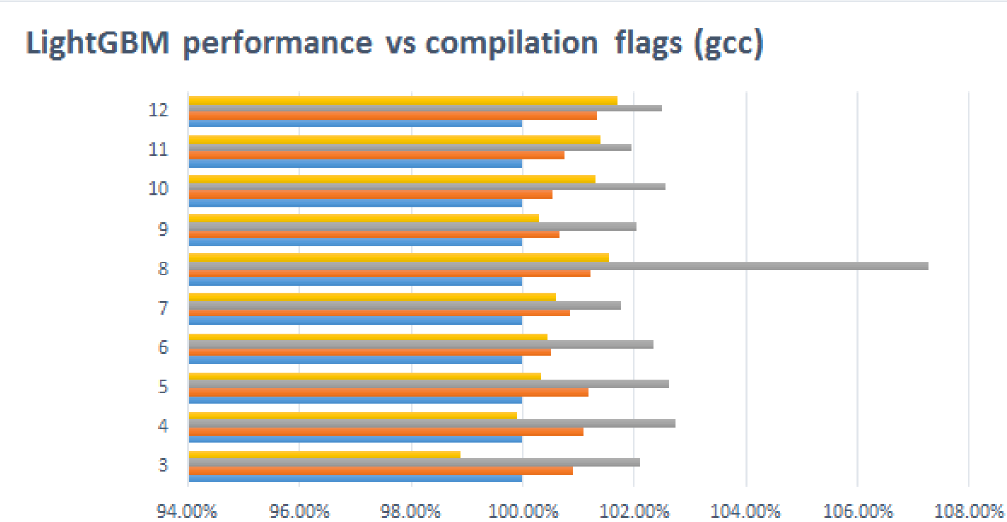
- Laurae++/Benchmarks
- Laurae2/gbt_benchmarks
- Laurae's Benchmark Master Data (Interactive)
- Kaggle Paris Meetup #12 Slides

Some explanatory pictures:

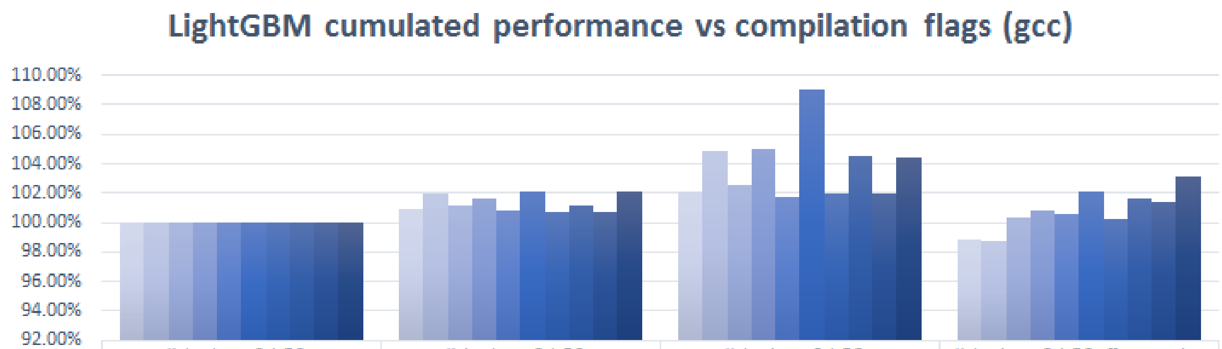
Parameter	Algorithm	Threads 1	2	3	4	5	6	12	Total
Depth: 03	lgb-v2	2,477	1,533	1,370	1,267	1,051	956	873	9,528
	lgb-v2-O2	2,438	1,521	1,358	1,261	1,035	965	862	9,440
	lgb-v2-O3	2,343	1,509	1,349	1,254	1,042	970	862	9,328
	lgb-v2-O3-fmath	2,412	1,588	1,390	1,321	1,076	976	872	9,634
Depth: 06	lgb-v2	1,851	1,183	1,039	970	821	788	730	7,381
	lgb-v2-O2	1,830	1,180	1,035	971	807	795	725	7,343
	lgb-v2-O3	1,745	1,155	1,025	959	819	783	723	7,208
	lgb-v2-O3-fmath	1,799	1,173	1,038	997	827	788	725	7,347
Depth: 10	lgb-v2	1,083	774	712	703	608	621	659	5,159
	lgb-v2-O2	1,091	772	708	689	595	625	651	5,132
	lgb-v2-O3	1,017	759	697	685	605	619	646	5,028
	lgb-v2-O3-fmath	1,055	762	704	704	603	616	647	5,091
Depth: 12	lgb-v2	1,237	913	851	853	749	770	855	6,228
	lgb-v2-O2	1,216	907	850	830	730	766	845	6,146
	lgb-v2-O3	1,159	894	835	824	753	773	835	6,073
	lgb-v2-O3-fmath	1,191	898	839	850	734	763	847	6,122



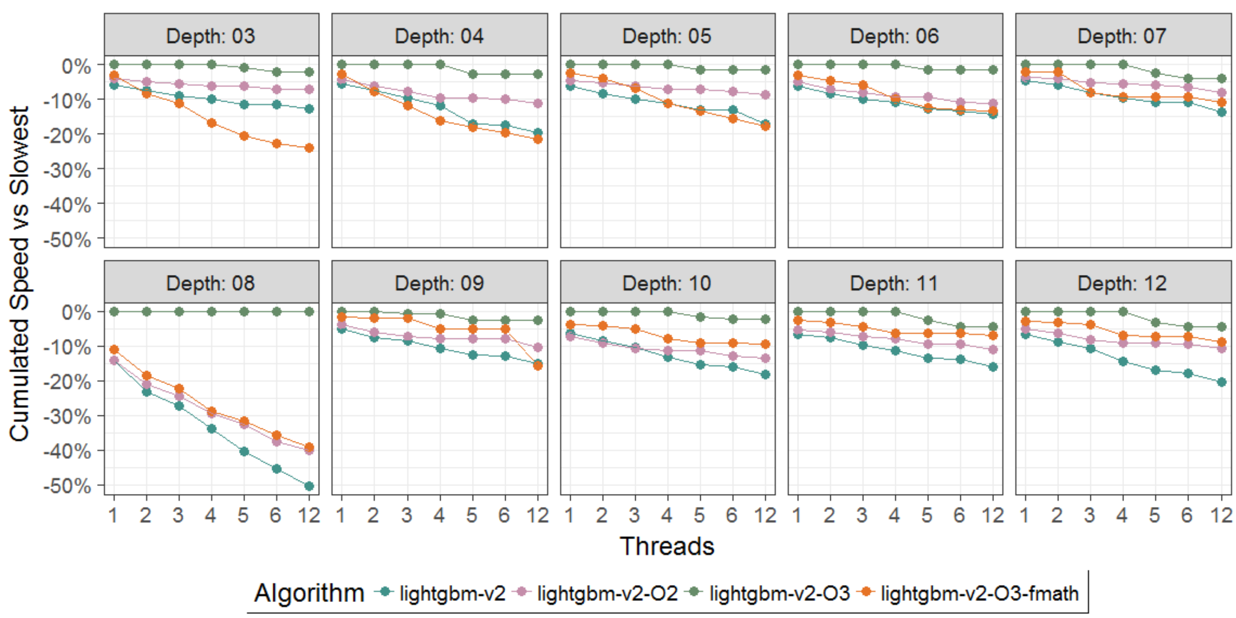


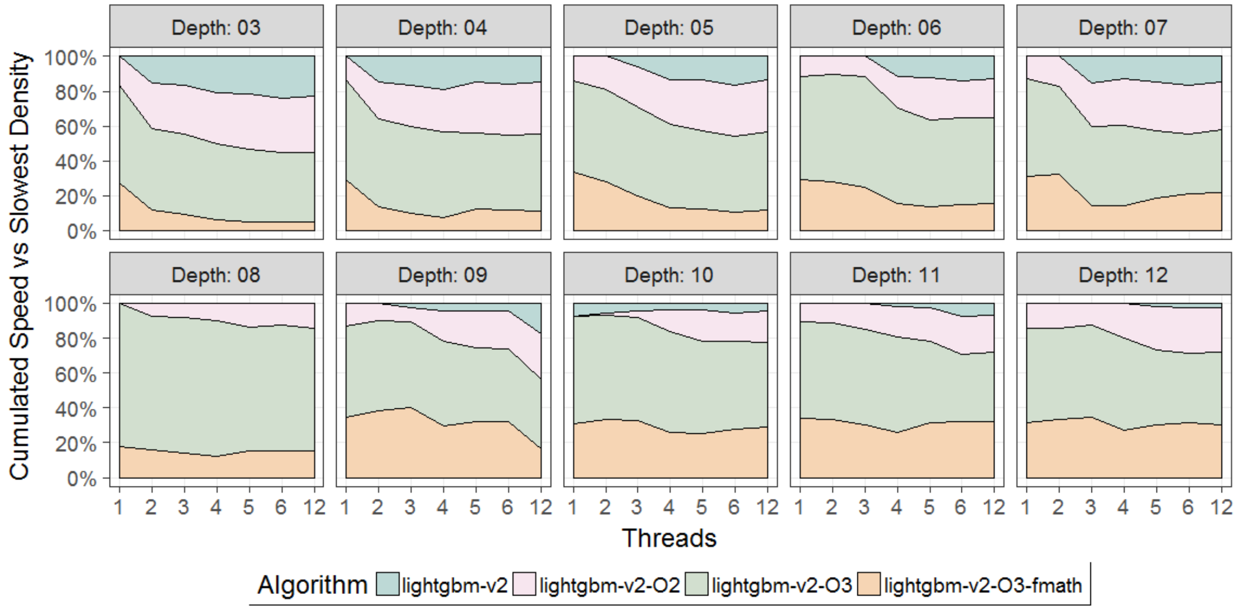


	3	4	5	6	7	8	9	10	11	12
lightgbm-v2 (-O3 -ffast-math -mtune=native)	98.88%	99.91%	100.33%	100.46%	100.59%	101.54%	100.28%	101.31%	101.38%	101.70%
lightgbm-v2 (-O3 -mtune=native)	102.10%	102.73%	102.60%	102.34%	101.75%	107.27%	102.03%	102.54%	101.95%	102.49%
lightgbm-v2 (-O2 -mtune=native)	100.91%	101.10%	101.17%	100.52%	100.85%	101.22%	100.66%	100.53%	100.74%	101.33%
lightgbm-v2 (-O2 -mtune=core2)	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%



	lightgbm-v2 (-O2 -mtune=core2)	lightgbm-v2 (-O2 -mtune=native)	lightgbm-v2 (-O3 -mtune=native)	lightgbm-v2 (-O3 -ffast-math -mtune=native)
3	100.00%	100.91%	102.10%	98.88%
4	100.00%	102.01%	104.83%	98.79%
5	100.00%	101.17%	102.60%	100.33%
6	100.00%	101.68%	104.94%	100.79%
7	100.00%	100.85%	101.75%	100.59%
8	100.00%	102.07%	109.02%	102.13%
9	100.00%	100.66%	102.03%	100.28%
10	100.00%	101.19%	104.57%	101.59%
11	100.00%	100.74%	101.95%	101.38%
12	100.00%	102.06%	104.45%	103.08%




Depth: 03
Depth: 04
Depth: 05
Depth: 06
Depth: 07
Depth: 08
Depth: 09
Depth: 10
Depth: 11
Depth: 12
Cumulated Speed vs Slowest Density
100%
80%
60%
40%
20%
0%
1 2 3 4 5 6 12
Threads
Algorithm
lightgbm-v2
lightgbm-v2-O2
lightgbm-v2-O3
lightgbm-v2-O3-fmath

CHAPTER 18

Documentation

Documentation for LightGBM is generated using Sphinx.

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.

18.1 Build

You can build the documentation locally. Just run in `docs` folder

for Python 3.x:

```
pip install sphinx "sphinx_rtd_theme>=0.3"
make html
```

for Python 2.x:

```
pip install mock sphinx "sphinx_rtd_theme>=0.3"
make html
```

CHAPTER 19

Indices and Tables

- genindex

Index

A

B

C

D

E

F

G

L

M

N

O

P

R

S

T

U