



LightGBM

Release 4.6.0.99

Microsoft Corporation

Jan 06, 2026

CONTENTS:

1	Installation Guide	3
2	Quick Start	21
3	Python-package Introduction	23
4	Features	29
5	Experiments	35
6	Parameters	39
7	Parameters Tuning	63
8	C API	69
9	Python API	105
10	Distributed Learning Guide	225
11	LightGBM GPU Tutorial	235
12	Advanced Topics	239
13	LightGBM FAQ	243
14	Development Guide	253
15	GPU Tuning Guide and Performance Comparison	255
16	GPU SDK Correspondence and Device Targeting Table	259
17	Documentation	263
18	Indices and Tables	265
Index		267

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

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

For more details, please refer to [Features](#).

INSTALLATION GUIDE

1.1 Versioning

LightGBM releases use a 3-part version number, with this format:

```
{major}.{minor}.{patch}
```

This version follows a scheme called Intended Effort Versioning (“Effver” for short). Changes to a component of the version indicate how much effort it will likely take to update code using a previous version.

- **major** = updating will require significant effort
- **minor** = some effort
- **patch** = no or very little effort

This means that **new minor versions can contain breaking changes**, but these are typically small or limited to less-frequently-used parts of the project.

When built from source on an unreleased commit, this version takes the following form:

```
{major}.{minor}.{patch}.99
```

That `.99` is added to ensure that a version built from an unreleased commit is considered “newer” than all previous releases, and “older” than all future releases.

To download such artifacts, run the following from the root of this repository.

```
bash .ci/download-artifacts.sh ${COMMIT_ID}
```

Where `COMMIT_ID` is the full commit SHA pointing to a commit on `master`. The artifacts can then be found in the `release-artifacts/` directory.

For more details on why LightGBM uses EffVer instead of other schemes like semantic versioning, see <https://jacobtomlinson.dev/effver/>.

1.2 General Installation Notes

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

By default, instructions below will use **VS Build Tools** or **make** tool to compile the code. It is possible to use **Ninja** tool instead of **make** on all platforms, but **VS Build Tools** cannot be replaced with **Ninja**. You can add `-G Ninja` to **CMake** flags to use **Ninja**.

By default, instructions below will produce a shared library file and an executable file with command-line interface. You can add `-DBUILD_CLI=OFF` to CMake flags to disable the executable compilation.

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

By default, instructions below will place header files into system-wide folder. You can add `-DINSTALL_HEADERS=OFF` to CMake flags to disable headers installation.

By default, on macOS, CMake is looking into Homebrew standard folders for finding dependencies (e.g. OpenMP). You can add `-DUSE_HOMEBREW_FALLBACK=OFF` to CMake flags to disable this behaviour.

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

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

In addition to the debug mode, LightGBM can be built with compiler sanitizers. To enable them add `-DUSE_SANITIZER=ON` `-DENABLED_SANITIZERS="address;leak;undefined"` to CMake flags. These values refer to the following supported sanitizers:

- `address` - AddressSanitizer (ASan);
- `leak` - LeakSanitizer (LSan);
- `undefined` - UndefinedBehaviorSanitizer (UBSan);
- `thread` - ThreadSanitizer (TSan).

Please note, that ThreadSanitizer cannot be used together with other sanitizers. For more info and additional sanitizers' parameters please refer to the [following docs](#). It is very useful to build [C++ unit tests](#) with sanitizers.

1.3 Windows

On Windows, LightGBM can be built using

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

1.3.1 Visual Studio (or VS Build Tools)

With GUI

1. Install [Visual Studio](#).
2. Navigate to one of the releases at <https://github.com/microsoft/LightGBM/releases>, download `LightGBM-complete_source_code_zip.zip`, and unzip it.
3. Go to `LightGBM-complete_source_code_zip/windows` folder.
4. Open `LightGBM.sln` file with **Visual Studio**, choose **Release** configuration if you need executable file or **DLL** configuration if you need shared library 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.

If you have errors about **Windows SDK Version**, go to **Project -> Properties -> Configuration Properties -> General** and select the SDK installed on your machine.

The .exe file will be in `LightGBM-complete_source_code_zip/windows/x64/Release` folder. The .dll file will be in `LightGBM-complete_source_code_zip/windows/x64/DLL` folder.

From Command Line

1. Install [Git for Windows](#), [CMake](#) and [VS Build Tools](#) ([VS Build Tools](#) is not needed if [Visual Studio](#) is already installed).
2. Run the following commands:

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

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

1.3.2 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
cmake -B build -S . -G "MinGW Makefiles"
cmake --build build -j4
```

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

Note: You may need to run the `cmake -B build -S . -G "MinGW Makefiles"` one more time or add `-DCMAKE_SH=CMAKE_SH-NOTFOUND` to CMake flags if you encounter the `sh.exe` was found in your PATH error.

It is recommended that you use [Visual Studio](#) since it has better multithreading efficiency in [Windows](#) for many-core systems (see [Question 4](#) and [Question 8](#)).

1.4 Linux

On Linux, LightGBM can be built using

- [CMake](#) and [gcc](#);
- [CMake](#) and [Clang](#).

After compilation the executable and .so files will be in `LightGBM/` folder.

1.4.1 gcc

1. Install [CMake](#) and [gcc](#).
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S .
cmake --build build -j4
```

1.4.2 Clang

1. Install **CMake**, **Clang** and **OpenMP**.
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=clang++-14 CC=clang-14 # replace "14" with version of Clang installed
on your machine
cmake -B build -S .
cmake --build build -j4
```

1.5 macOS

On macOS, LightGBM can be installed using

- **Homebrew**;
- **MacPorts**;

or can be built using

- **CMake** and **Apple Clang**;
- **CMake** and **gcc**.

1.5.1 Install Using Homebrew

```
brew install lightgbm
```

Refer to <https://formulae.brew.sh/formula/lightgbm> for more details.

1.5.2 Install Using MacPorts

```
sudo port install LightGBM
```

Refer to <https://ports.macports.org/port/LightGBM> for more details.

Note: Port for LightGBM is not maintained by LightGBM's maintainers.

1.5.3 Build from GitHub

After compilation the executable and `.dylib` files will be in `LightGBM/` folder.

Apple Clang

1. Install **CMake** and **OpenMP**:

```
brew install cmake libomp
```

2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S .
cmake --build build -j4
```

gcc

1. Install CMake and gcc:

```
brew install cmake gcc
```

2. 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
cmake -B build -S .
cmake --build build -j4
```

1.6 Docker

Refer to Docker folder.

1.7 Build Threadless Version (not Recommended)

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

1.7.1 Windows

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

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

Visual Studio (or VS Build Tools)

With GUI

1. Install Visual Studio.
2. Navigate to one of the releases at <https://github.com/microsoft/LightGBM/releases>, download `LightGBM-complete_source_code_zip.zip`, and unzip it.
3. Go to `LightGBM-complete_source_code_zip/windows` folder.
4. Open `LightGBM.sln` file with **Visual Studio**, choose Release configuration if you need executable file or DLL configuration if you need shared library.
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 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.

If you have errors about **Windows SDK Version**, go to Project -> Properties -> Configuration Properties -> General and select the SDK installed on your machine.

The .exe file will be in `LightGBM-complete_source_code_zip/windows/x64/Release` folder. The .dll file will be in `LightGBM-complete_source_code_zip/windows/x64/DLL` folder.

From Command Line

1. Install [Git for Windows](#), [CMake](#) and [VS Build Tools](#) ([VS Build Tools](#) is not needed if [Visual Studio](#) is already installed).
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S . -A x64 -DUSE_OPENMP=OFF
cmake --build 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
cmake -B build -S . -G "MinGW Makefiles" -DUSE_OPENMP=OFF
cmake --build build -j4
```

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

Note: You may need to run the `cmake -B build -S . -G "MinGW Makefiles" -DUSE_OPENMP=OFF` one more time or add `-DCMAKE_SH=CMAKE_SH-NOTFOUND` to CMake flags if you encounter the `sh.exe` was found in your PATH error.

1.7.2 Linux

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

- [CMake](#) and [gcc](#);
- [CMake](#) and [Clang](#).

After compilation the executable and .so files will be in `LightGBM/` folder.

gcc

1. Install [CMake](#) and [gcc](#).
2. Run the following commands:

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

Clang

1. Install [CMake](#) and [Clang](#).
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=clang++-14 CC=clang-14 # replace "14" with version of Clang installed
on your machine
cmake -B build -S . -DUSE_OPENMP=OFF
cmake --build build -j4
```

1.7.3 macOS

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

- [CMake](#) and [Apple Clang](#);
- [CMake](#) and [gcc](#).

After compilation the executable and .dylib files will be in LightGBM/ folder.

Apple Clang

1. Install [CMake](#):

```
brew install cmake
```

2. Run the following commands:

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

gcc

1. Install [CMake](#) and [gcc](#):

```
brew install cmake gcc
```

2. 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
cmake -B build -S . -DUSE_OPENMP=OFF
cmake --build build -j4
```

1.8 Build MPI Version

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

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

1.8.1 Windows

On Windows, an MPI version of LightGBM can be built using

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

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

With GUI

1. You need to install **MS MPI** first. Both `msmpisdk.msi` and `msmpisetup.exe` are needed.
2. Install **Visual Studio**.
3. Navigate to one of the releases at <https://github.com/microsoft/LightGBM/releases>, download `LightGBM-complete_source_code_zip.zip`, and unzip it.
4. Go to `LightGBM-complete_source_code_zip/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.

If you have errors about **Windows SDK Version**, go to `Project -> Properties -> Configuration Properties -> General` and select the SDK installed on your machine.

The `.exe` file will be in `LightGBM-complete_source_code_zip/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** and **VS Build Tools** (**VS Build Tools** is not needed if **Visual Studio** is already installed).
3. Run the following commands:

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

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

1.8.2 Linux

On Linux, an MPI version of LightGBM can be built using

- **CMake, gcc and Open MPI;**
- **CMake, Clang and Open MPI.**

After compilation the executable and `.so` files will be in `LightGBM/` folder.

gcc

1. Install **CMake**, **gcc** and **Open MPI**.
2. Run the following commands:

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

Clang

1. Install **CMake**, **Clang**, **OpenMP** and **Open MPI**.
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=clang++-14 CC=clang-14 # replace "14" with version of Clang installed
on your machine
cmake -B build -S . -DUSE_MPI=ON
cmake --build build -j4
```

1.8.3 macOS

On macOS, an MPI version of LightGBM can be built using

- **CMake**, **Open MPI** and **Apple Clang**;
- **CMake**, **Open MPI** and **gcc**.

After compilation the executable and .dylib files will be in LightGBM/ folder.

Apple Clang

1. Install **CMake**, **OpenMP** and **Open MPI**:

```
brew install cmake libomp open-mpi
```

2. Run the following commands:

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

gcc

1. Install **CMake**, **Open MPI** and **gcc**:

```
brew install cmake open-mpi gcc
```

2. 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
cmake -B build -S . -DUSE_MPI=ON
cmake --build build -j4
```

1.9 Build GPU Version

1.9.1 Windows

On Windows, a GPU version of LightGBM (device_type=gpu) can be built using

- **OpenCL, Boost, CMake and VS Build Tools;**
- **OpenCL, Boost, CMake and MinGW.**

If you use **MinGW**, the build procedure is similar to the build on Linux.

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

1. Install **Git for Windows**, **CMake** and **VS Build Tools** (**VS Build Tools** is not needed if **Visual Studio** is installed).
2. Install **OpenCL** for Windows. The installation depends on the brand (NVIDIA, AMD, Intel) of your GPU card.
 - For running on Intel, get **Intel SDK for OpenCL**.
 - For running on AMD, get **AMD APP SDK**.
 - For running on NVIDIA, get **CUDA Toolkit**.

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

3. Install **Boost Binaries**.

Note: Match your Visual C++ version:

Visual Studio 2017 -> **msvc-14.1-64.exe**,

Visual Studio 2019 -> **msvc-14.2-64.exe**,

Visual Studio 2022 -> **msvc-14.3-64.exe**.

4. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S . -A x64 -DUSE_GPU=ON -DBOOST_ROOT=C:/local/boost_1_63_0 -DBOOST_
                           ↪LIBRARYDIR=C:/local/boost_1_63_0/lib64-msvc-14.3
# if you have installed NVIDIA CUDA to a customized location, you should specify
                           ↪paths to OpenCL headers and library like the following:
# cmake -B build -S . -A x64 -DUSE_GPU=ON -DBOOST_ROOT=C:/local/boost_1_63_0 -
                           ↪DBOOST_LIBRARYDIR=C:/local/boost_1_63_0/lib64-msvc-14.3 -DOpenCL_LIBRARY="C:/-
                           ↪Program Files/NVIDIA GPU Computing Toolkit/CUDA/v10.0/lib/x64/OpenCL.lib" -
                           ↪DOpenCL_INCLUDE_DIR="C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v10.0/
                           ↪include"
cmake --build build --target ALL_BUILD --config Release
```

Note: **C:/local/boost_1_63_0** and **C:/local/boost_1_63_0/lib64-msvc-14.3** are locations of your **Boost** binaries (assuming you've downloaded 1.63.0 version for Visual Studio 2022).

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

1.9.2 Linux

On Linux, a GPU version of LightGBM (device_type=gpu) can be built using

- **CMake, OpenCL, Boost and gcc;**
- **CMake, OpenCL, Boost and Clang.**

OpenCL headers and libraries are usually provided by GPU manufacturer. The generic OpenCL ICD packages (for example, Debian packages `ocl-icd-libopencl1`, `ocl-icd-opencl-dev`, `polocl-opencl-icd`) can also be used.

Required **Boost** libraries (`Boost.Align`, `Boost.System`, `Boost.Filesystem`, `Boost.Chrono`) should be provided by the following Debian packages: `libboost-dev`, `libboost-system-dev`, `libboost-filesystem-dev`, `libboost-chrono-dev`.

After compilation the executable and .so files will be in LightGBM/ folder.

gcc

1. Install **CMake**, **gcc**, **OpenCL** and **Boost**.

2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S . -DUSE_GPU=ON
# if you have installed NVIDIA CUDA to a customized location, you should specify ↵
# paths to OpenCL headers and library like the following:
# cmake -B build -S . -DUSE_GPU=ON -DOpenCL_LIBRARY=/usr/local/cuda/lib64/libOpenCL. ↵
# so -DOpenCL_INCLUDE_DIR=/usr/local/cuda/include/
cmake --build build -j4
```

Clang

1. Install **CMake**, **Clang**, **OpenMP**, **OpenCL** and **Boost**.

2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=clang++-14 CC=clang-14 # replace "14" with version of Clang installed ↵
# on your machine
cmake -B build -S . -DUSE_GPU=ON
# if you have installed NVIDIA CUDA to a customized location, you should specify ↵
# paths to OpenCL headers and library like the following:
# cmake -B build -S . -DUSE_GPU=ON -DOpenCL_LIBRARY=/usr/local/cuda/lib64/libOpenCL. ↵
# so -DOpenCL_INCLUDE_DIR=/usr/local/cuda/include/
cmake --build build -j4
```

1.9.3 macOS

The GPU version is not supported on macOS.

1.9.4 Docker

Refer to [GPU Docker](#) folder.

1.10 Build CUDA Version

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

The CUDA-based version (device_type=cuda) is a separate implementation. Use this version in Linux environments with an NVIDIA GPU with compute capability 6.0 or higher.

1.10.1 Windows

The CUDA version is not supported on Windows. Use the *GPU version* (device_type=gpu) for GPU acceleration on Windows.

1.10.2 Linux

On Linux, a CUDA version of LightGBM can be built using

- **CMake, gcc and CUDA;**
- **CMake, Clang and CUDA.**

Please refer to [this detailed guide](#) for CUDA libraries installation.

After compilation the executable and .so files will be in `LightGBM/` folder.

gcc

1. Install **CMake, gcc and CUDA.**
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S . -DUSE_CUDA=ON
cmake --build build -j4
```

Clang

1. Install **CMake, Clang, OpenMP and CUDA.**
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=clang++-14 CC=clang-14 # replace "14" with version of Clang installed
on your machine
cmake -B build -S . -DUSE_CUDA=ON
cmake --build build -j4
```

1.10.3 macOS

The CUDA version is not supported on macOS.

1.11 Build Java Wrapper

Using the following instructions you can generate a JAR file containing the LightGBM [C API](#) wrapped by **SWIG**.

After compilation the `.jar` file will be in `LightGBM/build` folder.

1.11.1 Windows

On Windows, a Java wrapper of LightGBM can be built using

- **Java, SWIG, CMake and VS Build Tools;**
- **Java, SWIG, CMake and MinGW.**

VS Build Tools

1. Install Git for Windows, [CMake](#) and [VS Build Tools](#) (**VS Build Tools** is not needed if **Visual Studio** is already installed).
2. Install [SWIG](#) and **Java** (also make sure that `JAVA_HOME` environment variable is set properly).
3. Run the following commands:

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

MinGW-w64

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

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S . -G "MinGW Makefiles" -DUSE_SWIG=ON
cmake --build build -j4
```

Note: You may need to run the `cmake -B build -S . -G "MinGW Makefiles" -DUSE_SWIG=ON` one more time or add `-DCMAKE_SH=CMAKE_SH-NOTFOUND` to CMake flags if you encounter the `sh.exe` was found in your PATH error.

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

1.11.2 Linux

On Linux, a Java wrapper of LightGBM can be built using

- **CMake, gcc, Java and SWIG;**
- **CMake, Clang, Java and SWIG.**

gcc

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

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

Clang

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

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=clang++-14 CC=clang-14 # replace "14" with version of Clang installed
on your machine
cmake -B build -S . -DUSE_SWIG=ON
cmake --build build -j4
```

1.11.3 macOS

On macOS, a Java wrapper of LightGBM can be built using

- **CMake**, **Java**, **SWIG** and **Apple Clang**;
- **CMake**, **Java**, **SWIG** and **gcc**.

Apple Clang

1. Install **CMake**, **Java** (also make sure that **JAVA_HOME** environment variable is set properly), **SWIG** and **OpenMP**:

```
brew install cmake openjdk swig libomp
export JAVA_HOME="$(brew --prefix openjdk)/libexec/openjdk.jdk/Contents/Home/"
```

2. Run the following commands:

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

gcc

1. Install **CMake**, **Java** (also make sure that **JAVA_HOME** environment variable is set properly), **SWIG** and **gcc**:

```
brew install cmake openjdk swig gcc
export JAVA_HOME="$(brew --prefix openjdk)/libexec/openjdk.jdk/Contents/Home/"
```

2. 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
cmake -B build -S . -DUSE_SWIG=ON
cmake --build build -j4
```

1.12 Build Python-package

Refer to [Python-package](#) folder.

1.13 Build R-package

Refer to [R-package](#) folder.

1.14 Build C++ Unit Tests

1.14.1 Windows

On Windows, C++ unit tests of LightGBM can be built using

- [CMake and VS Build Tools](#);
- [CMake and MinGW](#).

VS Build Tools

1. Install [Git for Windows](#), [CMake](#) and [VS Build Tools](#) (**VS Build Tools** is not needed if **Visual Studio** is already installed).
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S . -A x64 -DBUILD_CPP_TEST=ON
cmake --build build --target testlightgbm --config Debug
```

The .exe file will be in `LightGBM/Debug` 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
cmake -B build -S . -G "MinGW Makefiles" -DBUILD_CPP_TEST=ON
cmake --build build --target testlightgbm -j4
```

The .exe file will be in `LightGBM/` folder.

Note: You may need to run the `cmake -B build -S . -G "MinGW Makefiles" -DBUILD_CPP_TEST=ON` one more time or add `-DCMAKE_SH=CMAKE_SH-NOTFOUND` to CMake flags if you encounter the `sh.exe` was found in your PATH error.

1.14.2 Linux

On Linux, a C++ unit tests of LightGBM can be built using

- **CMake** and **gcc**;
- **CMake** and **Clang**.

After compilation the executable file will be in LightGBM/ folder.

gcc

1. Install **CMake** and **gcc**.
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S . -DBUILD_CPP_TEST=ON
cmake --build build --target testlightgbm -j4
```

Clang

1. Install **CMake**, **Clang** and **OpenMP**.
2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
export CXX=clang++-14 CC=clang-14 # replace "14" with version of Clang installed
on your machine
cmake -B build -S . -DBUILD_CPP_TEST=ON
cmake --build build --target testlightgbm -j4
```

1.14.3 macOS

On macOS, a C++ unit tests of LightGBM can be built using

- **CMake** and **Apple Clang**;
- **CMake** and **gcc**.

After compilation the executable file will be in LightGBM/ folder.

Apple Clang

1. Install **CMake** and **OpenMP**:

```
brew install cmake libomp
```

2. Run the following commands:

```
git clone --recursive https://github.com/microsoft/LightGBM
cd LightGBM
cmake -B build -S . -DBUILD_CPP_TEST=ON
cmake --build build --target testlightgbm -j4
```

gcc

1. Install **CMake** and **gcc**:

```
brew install cmake gcc
```

2. 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
cmake -B build -S . -DBUILD_CPP_TEST=ON
cmake --build build --target testlightgbm -j4
```


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](#) (zero-based) formats.

Files could be both with and without [headers](#).

[Label column](#) could be specified both by index and by name.

Some columns could be [ignored](#).

2.1.1 Categorical Feature Support

LightGBM can use categorical features directly (without one-hot encoding). The experiment on [Expo data](#) shows about 8x speed-up compared with one-hot encoding.

For the setting details, please refer to the [categorical_feature](#) parameter.

2.1.2 Weight and Query/Group Data

LightGBM also supports weighted training, it needs an additional [weight](#) data. And it needs an additional [query](#) data for ranking task.

Also, [weight](#) and [query](#) data could be specified as columns in training data in the same manner as label.

2.2 Parameters Quick Look

The parameters format is `key1=value1 key2=value2 ...`

Parameters can be set both in config file and command line. If one parameter appears in both command line and config file, LightGBM will use the parameter from the command line.

The most important parameters which new users should take a look at are located into [Core Parameters](#) and the top of [Learning Control Parameters](#) sections of the full detailed list of LightGBM's parameters.

2.3 Run LightGBM

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

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

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

2.4 Examples

- Binary Classification
- Regression
- Lambdarank
- Distributed Learning

PYTHON-PACKAGE INTRODUCTION

This document gives a basic walk-through of LightGBM Python-package.

List of other helpful links

- [Python Examples](#)
- [Python API](#)
- [Parameters Tuning](#)

3.1 Install

The preferred way to install LightGBM is via pip:

```
pip install lightgbm
```

Refer to [Python-package](#) folder for the detailed installation guide.

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

```
import lightgbm as lgb
```

3.2 Data Interface

The LightGBM Python module can load data from:

- LibSVM (zero-based) / TSV / CSV format text file
- NumPy 2D array(s), pandas DataFrame, pyarrow Table, SciPy sparse matrix
- LightGBM binary file
- LightGBM Sequence object(s)

The data is stored in a `Dataset` object.

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

```
import numpy as np
```

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

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

To load a numpy array into Dataset:

```
rng = np.random.default_rng()
data = rng.uniform(size=(500, 10)) # 500 entities, each contains 10 features
label = rng.integers(low=0, high=2, size=(500, )) # binary target
train_data = lgb.Dataset(data, label=label)
```

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

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

Load from Sequence objects:

We can implement Sequence interface to read binary files. The following example shows reading HDF5 file with h5py.

```
import h5py

class HDFSequence(lgb.Sequence):
    def __init__(self, hdf_dataset, batch_size):
        self.data = hdf_dataset
        self.batch_size = batch_size

    def __getitem__(self, idx):
        return self.data[idx]

    def __len__(self):
        return len(self.data)

f = h5py.File('train.hdf5', 'r')
train_data = lgb.Dataset(HDFSequence(f['X'], 8192), label=f['Y'][:])
```

Features of using Sequence interface:

- Data sampling uses random access, thus does not go through the whole dataset
- Reading data in batch, thus saves memory when constructing Dataset object
- Supports creating Dataset from multiple data files

Please refer to Sequence [API doc](#).

`dataset_from_multi_hdf5.py` is a detailed example.

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

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

Weights can be set when needed:

```
rng = np.random.default_rng()
w = rng.uniform(size=(500, ))
train_data = lgb.Dataset(data, label=label, weight=w)
```

or

```
train_data = lgb.Dataset(data, label=label)
rng = np.random.default_rng()
w = rng.uniform(size=(500, ))
train_data.set_weight(w)
```

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

Memory efficient usage:

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

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

3.3 Setting Parameters

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

- Booster parameters:

```
param = {'num_leaves': 31, '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:

```
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, callbacks=[lgb.
    early_stopping(stopping_rounds=5)])
bst.save_model('model.txt', num_iteration=bst.best_iteration)
```

The model will train until the validation score stops improving. Validation score needs to improve at least every `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` callback. 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. However, you can change this behavior and make LightGBM check only the first metric for early stopping by passing `first_metric_only=True` in `early_stopping` callback constructor.

3.7 Prediction

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

```
# 7 entities, each contains 10 features
rng = np.random.default_rng()
data = rng.uniform(size=(7, 10))
y_pred = 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)
```


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 distributed learning**

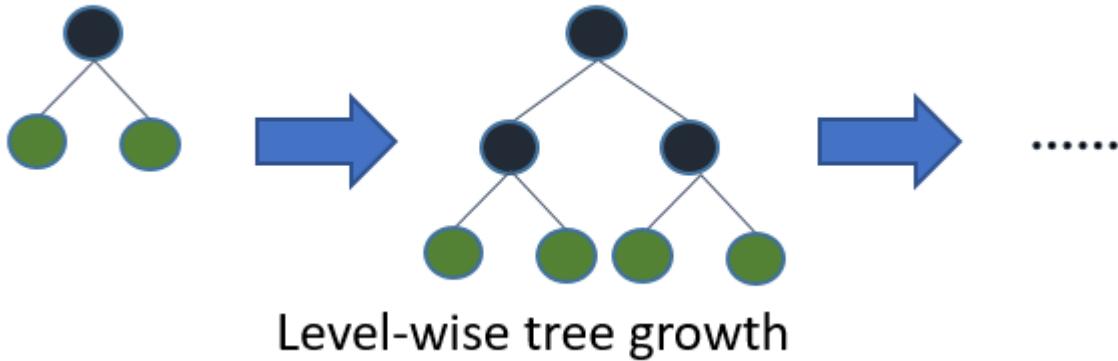
4.2 Sparse Optimization

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

4.3 Optimization in Accuracy

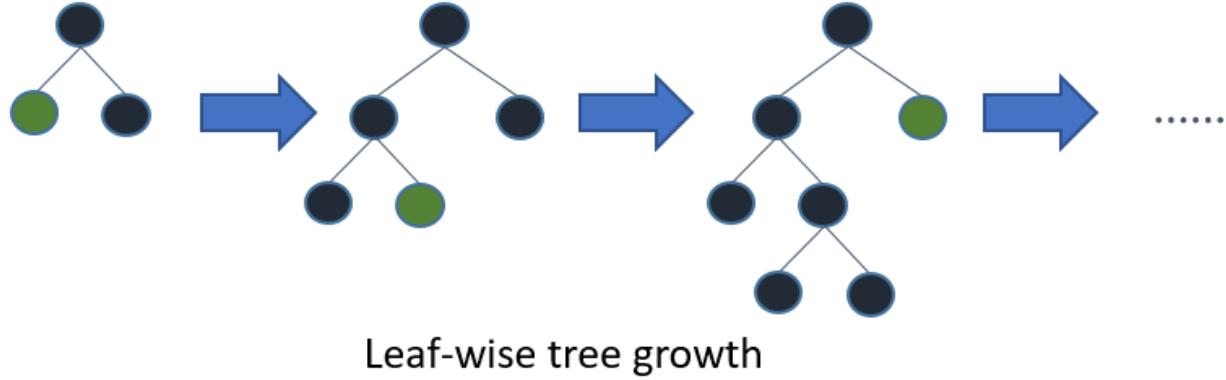
4.3.1 Leaf-wise (Best-first) Tree Growth

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 distributed learning of LightGBM. LightGBM implements state-of-the-art algorithms[9]. These collective communication algorithms can provide much better performance than point-to-point communication.

4.5 Optimization in Distributed Learning

LightGBM provides the following distributed learning algorithms.

4.5.1 Feature Parallel

Traditional Algorithm

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

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

The shortcomings of traditional feature parallel:

- Has computation overhead, since it cannot speed up “split”, whose time complexity is $O(\#data)$. 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 uses “Reduce Scatter” to merge histograms of different (non-overlapping) features for different workers. Then workers find the local best split on local merged histograms and sync up the global best split.
2. As aforementioned, LightGBM uses histogram subtraction to speed up training. Based on this, we can communicate histograms only for one leaf, and get its neighbor’s histograms by subtraction as well.

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

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
- AUC-mu (new in v3.0.0)
- Average precision (new in v3.1.0)
- 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
- Multiple validation data
- Multiple metrics
- Early stopping (both training and prediction)
- Prediction for leaf index

For more details, please refer to [Parameters](#).

4.9 References

[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.

EXPERIMENTS

5.1 Comparison Experiment

For the detailed experiment scripts and output logs, please refer to this [repo](#).

5.1.1 History

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

27 Feb, 2017: first version.

5.1.2 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

5.1.3 Environment

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

OS	CPU	Memory
Ubuntu 16.04 LTS	2 * E5-2690 v4	448GB

5.1.4 Baseline

We used `xgboost` as a baseline.

Both `xgboost` and LightGBM were built with OpenMP support.

5.1.5 Settings

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

1. `xgboost`:

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

2. `xgboost_hist` (using histogram based algorithm):

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

3. LightGBM:

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

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

Other parameters are default values.

5.1.6 Result

Speed

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

The following table is the comparison of time cost:

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

LightGBM ran faster than xgboost on all experiment data sets.

Accuracy

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

Data	Metric	xgboost	xgboost_hist	LightGBM
Higgs	AUC	0.839593	0.845314	0.845724
Yahoo LTR	ND _C _G ₁	0.719748	0.720049	0.732981
	ND _C _G ₃	0.717813	0.722573	0.735689
	ND _C _G ₅	0.737849	0.740899	0.75352
	ND _C _G ₁₀	0.78089	0.782957	0.793498
MS LTR	ND _C _G ₁	0.483956	0.485115	0.517767
	ND _C _G ₃	0.467951	0.47313	0.501063
	ND _C _G ₅	0.472476	0.476375	0.504648
	ND _C _G ₁₀	0.492429	0.496553	0.524252
Expo	AUC	0.756713	0.776224	0.776935
Allstate	AUC	0.607201	0.609465	0.609072

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 (col-wise)	LightGBM (row-wise)
Higgs	4.853GB	7.335GB	0.897GB	1.401GB
Yahoo LTR	1.907GB	4.023GB	1.741GB	2.161GB
MS LTR	5.469GB	7.491GB	0.940GB	1.296GB
Expo	1.553GB	2.606GB	0.555GB	0.711GB
Allstate	6.237GB	12.090GB	1.116GB	1.755GB

5.2 Parallel Experiment

5.2.1 History

27 Feb, 2017: first version.

5.2.2 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 click-through 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.

5.2.3 Environment

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

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

5.2.4 Settings

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

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

5.2.5 Results

#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 distributed learning.

5.3 GPU Experiments

Refer to [GPU Performance](#).

PARAMETERS

This page contains descriptions of all parameters in LightGBM.

List of other helpful links

- [Python API](#)
- [Parameters Tuning](#)

6.1 Parameters Format

Parameters are merged together in the following order (later items overwrite earlier ones):

1. LightGBM's default values
2. special files for `weight`, `init_score`, `query`, and `positions` (see [Others](#))
3. (CLI only) configuration in a file passed like `config=train.conf`
4. (CLI only) configuration passed via the command line
5. (Python, R) special keyword arguments to some functions (e.g. `num_boost_round` in `train()`)
6. (Python, R) `params` function argument (including `**kwargs` in Python and `...` in R)
7. (C API) `parameters` or `params` function argument

Many parameters have “aliases”, alternative names which refer to the same configuration.

Where a mix of the primary parameter name and aliases are given, the primary parameter name is always preferred to any aliases.

For example, in Python:

```
# use learning rate of 0.07, because 'learning_rate'  
# is the primary parameter name  
lgb.train(  
    params={  
        "learning_rate": 0.07,  
        "shrinkage_rate": 0.12  
    },  
    train_set=dtrain  
)
```

Where multiple aliases are given, and the primary parameter name is not, the first alias appearing in the lists returned by `Config::parameter2aliases()` in the C++ library is used. Those lists are hard-coded in a fairly arbitrary way... wherever possible, avoid relying on this behavior.

For example, in Python:

```
# use learning rate of 0.12, LightGBM has a hard-coded preference for 'shrinkage_rate'  
# over any other aliases, and 'learning_rate' is not provided  
lgb.train(  
    params={  
        "eta": 0.19,  
        "shrinkage_rate": 0.12  
    },  
    train_set=dtrain  
)
```

CLI

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.

Python

Any parameters that accept multiple values should be passed as a Python list.

```
params = {  
    "monotone_constraints": [-1, 0, 1]  
}
```

R

Any parameters that accept multiple values should be passed as an R list.

```
params <- list(  
    monotone_constraints = c(-1, 0, 1)  
)
```

6.2 Core Parameters

- `config`, default = "", type = string, aliases: `config_file`
 - path of config file
 - **Note:** can be used only in CLI version
- `task`, default = `train`, type = enum, options: `train`, `predict`, `convert_model`, `refit`, aliases: `task_type`
 - `train`, for training, aliases: `training`
 - `predict`, for prediction, aliases: `prediction`, `test`
 - `convert_model`, for converting model file into if-else format, see more information in [Convert Parameters](#)
 - `refit`, for refitting existing models with new data, aliases: `refit_tree`
 - `save_binary`, load train (and validation) data then save dataset to binary file. Typical usage: `save_binary` first, then run multiple `train` tasks in parallel using the saved binary file
 - **Note:** can be used only in CLI version; for language-specific packages you can use the correspondent functions
- `objective`, default = `regression`, type = enum, options: `regression`, `regression_l1`, `huber`, `fair`, `poisson`, `quantile`, `mape`, `gamma`, `tweedie`, `binary`, `multiclass`, `multiclassova`, `cross_entropy`, `cross_entropy_lambda`, `lambdarank`, `rank_xendcg`, aliases: `objective_type`, `app`, `application`, `loss`

- regression application
 - * `regression`, L2 loss, aliases: `regression_l2`, `l2`, `mean_squared_error`, `mse`, `l2_root`, `root_mean_squared_error`, `rmse`
 - * `regression_l1`, L1 loss, aliases: `l1`, `mean_absolute_error`, `mae`
 - * `huber`, Huber loss
 - * `fair`, Fair loss
 - * `poisson`, Poisson regression
 - * `quantile`, Quantile regression
 - * `mape`, MAPE loss, aliases: `mean_absolute_percentage_error`
 - * `gamma`, Gamma regression with log-link. It might be useful, e.g., for modeling insurance claims severity, or for any target that might be `gamma-distributed`
 - * `tweedie`, Tweedie regression with log-link. It might be useful, e.g., for modeling total loss in insurance, or for any target that might be `tweedie-distributed`
- binary classification application
 - * `binary`, binary `log loss` classification (or logistic regression)
 - * requires labels in {0, 1}; see `cross-entropy` application for general probability labels in [0, 1]
- multi-class classification application
 - * `multiclass`, `softmax` objective function, aliases: `softmax`
 - * `multiclassova`, One-vs-All binary objective function, aliases: `multiclass_ova`, `ova`, `ovr`
 - * `num_class` should be set as well
- cross-entropy application
 - * `cross_entropy`, objective function for cross-entropy (with optional linear weights), aliases: `xentropy`
 - * `cross_entropy_lambda`, alternative parameterization of cross-entropy, aliases: `xentlambda`
 - * label is anything in interval [0, 1]
- ranking application
 - * `lambdarank`, `lambdarank` objective. `label_gain` can be used to set the gain (weight) of `int` label and all values in `label` must be smaller than number of elements in `label_gain`
 - * `rank_xendcg`, `XE_NDCG_MART` ranking objective function, aliases: `xendcg`, `xe_ndcg`, `xe_ndcg_mart`, `xendcg_mart`
 - * `rank_xendcg` is faster than and achieves the similar performance as `lambdarank`
 - * label should be `int` type, and larger number represents the higher relevance (e.g. 0:bad, 1:fair, 2:good, 3:perfect)
- custom objective function (gradients and hessians not computed directly by LightGBM)
 - * `custom`
 - * must be passed through parameters explicitly in the C API
 - * **Note:** cannot be used in CLI version
- `boosting`, default = `gbdt`, type = enum, options: `gbdt`, `rf`, `dart`, 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
 - * **Note:** internally, LightGBM uses `gbdt` mode for the first `1 / learning_rate` iterations
- `data_sample_strategy`, default = `bagging`, type = enum, options: `bagging`, `goss`
 - `bagging`, Randomly Bagging Sampling
 - * **Note:** `bagging` is only effective when `bagging_freq > 0` and `bagging_fraction < 1.0`
 - `goss`, Gradient-based One-Side Sampling
 - *New in version 4.0.0*
- `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`, `nrounds`, `num_boost_round`, `n_estimators`, `max_iter`, 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`, `max_leaf_nodes`, constraints: `1 < num_leaves <= 131072`
 - max number of leaves in one tree
- `tree_learner`, default = `serial`, type = enum, options: `serial`, `feature`, `data`, `voting`, aliases: `tree`, `tree_type`, `tree_learner_type`
 - `serial`, single machine tree learner
 - `feature`, feature parallel tree learner, aliases: `feature_parallel`
 - `data`, data parallel tree learner, aliases: `data_parallel`
 - `voting`, voting parallel tree learner, aliases: `voting_parallel`
 - refer to [Distributed Learning Guide](#) to get more details
- `num_threads`, default = 0, type = int, aliases: `num_thread`, `nthread`, `nthreads`, `n_jobs`

- used only in `train`, `prediction` and `refit` tasks or in correspondent functions of language-specific packages
- number of threads for LightGBM
- `0` means default number of threads in OpenMP
- for the best speed, set this to the number of **real CPU cores**, not the number of threads (most CPUs use [hyper-threading](#) to generate 2 threads per CPU core)
- do not set it too large if your dataset is small (for instance, do not use 64 threads for a dataset with 10,000 rows)
- be aware a task manager or any similar CPU monitoring tool might report that cores not being fully utilized. **This is normal**
- for distributed learning, do not use all CPU cores because this will cause poor performance for the network communication
- **Note:** please **don't** change this during training, especially when running multiple jobs simultaneously by external packages, otherwise it may cause undesirable errors
- `device_type`, default = `cpu`, type = enum, options: `cpu`, `gpu`, `cuda`, aliases: `device`
 - device for the tree learning
 - `cpu` supports all LightGBM functionality and is portable across the widest range of operating systems and hardware
 - `cuda` offers faster training than `gpu` or `cpu`, but only works on GPUs supporting CUDA
 - `gpu` can be faster than `cpu` and works on a wider range of GPUs than CUDA
 - **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 or CUDA support
- `seed`, default = `None`, type = int, aliases: `random_seed`, `random_state`
 - this seed is used to generate other seeds, e.g. `data_random_seed`, `feature_fraction_seed`, etc.
 - by default, this seed is unused in favor of default values of other seeds
 - this seed has lower priority in comparison with other seeds, which means that it will be overridden, if you set other seeds explicitly
- `deterministic`, default = `false`, type = bool
 - used only with `cpu` device type
 - setting this to `true` should ensure the stable results when using the same data and the same parameters (and different `num_threads`)
 - when you use the different seeds, different LightGBM versions, the binaries compiled by different compilers, or in different systems, the results are expected to be different
 - you can [raise issues](#) in LightGBM GitHub repo when you meet the unstable results
 - **Note:** setting this to `true` may slow down the training
 - **Note:** to avoid potential instability due to numerical issues, please set `force_col_wise=true` or `force_row_wise=true` when setting `deterministic=true`

6.3 Learning Control Parameters

- `force_col_wise`, default = `false`, type = `bool`
 - used only with `cpu` device type
 - set this to `true` to force col-wise histogram building
 - enabling this is recommended when:
 - * the number of columns is large, or the total number of bins is large
 - * `num_threads` is large, e.g. > 20
 - * you want to reduce memory cost
 - **Note:** when both `force_col_wise` and `force_row_wise` are `false`, LightGBM will firstly try them both, and then use the faster one. To remove the overhead of testing set the faster one to `true` manually
 - **Note:** this parameter cannot be used at the same time with `force_row_wise`, choose only one of them
- `force_row_wise`, default = `false`, type = `bool`
 - used only with `cpu` device type
 - set this to `true` to force row-wise histogram building
 - enabling this is recommended when:
 - * the number of data points is large, and the total number of bins is relatively small
 - * `num_threads` is relatively small, e.g. $<= 16$
 - * you want to use small `bagging_fraction` or `goss` sample strategy to speed up
 - **Note:** setting this to `true` will double the memory cost for `Dataset` object. If you have not enough memory, you can try setting `force_col_wise=true`
 - **Note:** when both `force_col_wise` and `force_row_wise` are `false`, LightGBM will firstly try them both, and then use the faster one. To remove the overhead of testing set the faster one to `true` manually
 - **Note:** this parameter cannot be used at the same time with `force_col_wise`, choose only one of them
- `histogram_pool_size`, default = `-1.0`, type = `double`, aliases: `hist_pool_size`
 - max cache size in MB for historical histogram
 - < 0 means no limit
- `max_depth`, default = `-1`, type = `int`
 - limit the max depth for tree model. This is used to deal with over-fitting when `#data` is small. Tree still grows leaf-wise
 - $<= 0$ means no limit
- `min_data_in_leaf`, default = `20`, type = `int`, aliases: `min_data_per_leaf`, `min_data`, `min_child_samples`, `min_samples_leaf`, constraints: `min_data_in_leaf >= 0`
 - minimal number of data in one leaf. Can be used to deal with over-fitting
 - **Note:** this is an approximation based on the Hessian, so occasionally you may observe splits which produce leaf nodes that have less than this many observations
- `min_sum_hessian_in_leaf`, default = `1e-3`, type = `double`, aliases: `min_sum_hessian_per_leaf`, `min_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: $0.0 < \text{bagging_fraction} \leq 1.0$
 - like `feature_fraction`, but this will randomly select part of data without resampling
 - can be used to speed up training
 - can be used to deal with over-fitting
 - **Note:** to enable bagging, `bagging_freq` should be set to a non zero value as well
- `pos_bagging_fraction`, default = `1.0`, type = double, aliases: `pos_sub_row`, `pos_subsample`, `pos_bagging`, constraints: $0.0 < \text{pos_bagging_fraction} \leq 1.0$
 - used only in binary application
 - used for imbalanced binary classification problem, will randomly sample `#pos_samples * pos_bagging_fraction` positive samples in bagging
 - should be used together with `neg_bagging_fraction`
 - set this to `1.0` to disable
 - **Note:** to enable this, you need to set `bagging_freq` and `neg_bagging_fraction` as well
 - **Note:** if both `pos_bagging_fraction` and `neg_bagging_fraction` are set to `1.0`, balanced bagging is disabled
 - **Note:** if balanced bagging is enabled, `bagging_fraction` will be ignored
- `neg_bagging_fraction`, default = `1.0`, type = double, aliases: `neg_sub_row`, `neg_subsample`, `neg_bagging`, constraints: $0.0 < \text{neg_bagging_fraction} \leq 1.0$
 - used only in binary application
 - used for imbalanced binary classification problem, will randomly sample `#neg_samples * neg_bagging_fraction` negative samples in bagging
 - should be used together with `pos_bagging_fraction`
 - set this to `1.0` to disable
 - **Note:** to enable this, you need to set `bagging_freq` and `pos_bagging_fraction` as well
 - **Note:** if both `pos_bagging_fraction` and `neg_bagging_fraction` are set to `1.0`, balanced bagging is disabled
 - **Note:** if balanced bagging is enabled, `bagging_fraction` will be ignored
- `bagging_freq`, default = `0`, type = int, aliases: `subsample_freq`
 - frequency for bagging
 - `0` means disable bagging; `k` means perform bagging at every `k` iteration. Every `k`-th iteration, LightGBM will randomly select `bagging_fraction * 100%` of the data to use for the next `k` iterations
 - **Note:** bagging is only effective when $0.0 < \text{bagging_fraction} < 1.0$
- `bagging_seed`, default = `3`, type = int, aliases: `bagging_fraction_seed`
 - random seed for bagging
- `bagging_by_query`, default = `false`, type = bool
 - whether to do bagging sample by query
 - *New in version 4.6.0*

- **feature_fraction**, default = 1.0, type = double, aliases: `sub_feature`, `colsample_bytree`, constraints: $0.0 < \text{feature_fraction} \leq 1.0$
 - LightGBM will randomly select a subset of features on each iteration (tree) if `feature_fraction` is smaller than 1.0. For example, if you set it to 0.8, LightGBM will select 80% of features before training each tree
 - can be used to speed up training
 - can be used to deal with over-fitting
- **feature_fraction_bynode**, default = 1.0, type = double, aliases: `sub_feature_bynode`, `colsample_bynode`, constraints: $0.0 < \text{feature_fraction_bynode} \leq 1.0$
 - LightGBM will randomly select a subset of features on each tree node if `feature_fraction_bynode` is smaller than 1.0. For example, if you set it to 0.8, LightGBM will select 80% of features at each tree node
 - can be used to deal with over-fitting
 - **Note:** unlike `feature_fraction`, this cannot speed up training
 - **Note:** if both `feature_fraction` and `feature_fraction_bynode` are smaller than 1.0, the final fraction of each node is `feature_fraction * feature_fraction_bynode`
- **feature_fraction_seed**, default = 2, type = int
 - random seed for `feature_fraction`
- **extra_trees**, default = `false`, type = bool, aliases: `extra_tree`
 - use extremely randomized trees
 - if set to `true`, when evaluating node splits LightGBM will check only one randomly-chosen threshold for each feature
 - can be used to speed up training
 - can be used to deal with over-fitting
- **extra_seed**, default = 6, type = int
 - random seed for selecting thresholds when `extra_trees` is true
- **early_stopping_round**, default = 0, type = int, aliases: `early_stopping_rounds`, `early_stopping`, `n_iter_no_change`
 - will stop training if one metric of one validation data doesn't improve in last `early_stopping_round` rounds
 - ≤ 0 means disable
 - can be used to speed up training
- **early_stopping_min_delta**, default = 0.0, type = double, constraints: $\text{early_stopping_min_delta} \geq 0.0$
 - when early stopping is used (i.e. `early_stopping_round > 0`), require the early stopping metric to improve by at least this delta to be considered an improvement
 - *New in version 4.4.0*
- **first_metric_only**, default = `false`, type = bool
 - LightGBM allows you to provide multiple evaluation metrics. Set this to `true`, if you want to use only the first metric for early stopping
- **max_delta_step**, default = 0.0, type = double, aliases: `max_tree_output`, `max_leaf_output`

- used to limit the max output of tree leaves
- ≤ 0 means no constraint
- the final max output of leaves is `learning_rate * max_delta_step`
- `lambda_11`, default = `0.0`, type = double, aliases: `reg_alpha`, `l1_regularization`, constraints: `lambda_11 >= 0.0`
 - L1 regularization
- `lambda_12`, default = `0.0`, type = double, aliases: `reg_lambda`, `lambda`, `l2_regularization`, constraints: `lambda_12 >= 0.0`
 - L2 regularization
- `linear_lambda`, default = `0.0`, type = double, constraints: `linear_lambda >= 0.0`
 - linear tree regularization, corresponds to the parameter `lambda` in Eq. 3 of [Gradient Boosting with Piece-Wise Linear Regression Trees](#)
- `min_gain_to_split`, default = `0.0`, type = double, aliases: `min_split_gain`, constraints: `min_gain_to_split >= 0.0`
 - the minimal gain to perform split
 - can be used to speed up training
- `drop_rate`, default = `0.1`, type = double, aliases: `rate_drop`, constraints: `0.0 <= drop_rate <= 1.0`
 - used only in `dart`
 - dropout rate: a fraction of previous trees to drop during the dropout
- `max_drop`, default = `50`, type = int
 - used only in `dart`
 - max number of dropped trees during one boosting iteration
 - ≤ 0 means no limit
- `skip_drop`, default = `0.5`, type = double, constraints: `0.0 <= skip_drop <= 1.0`
 - used only in `dart`
 - probability of skipping the dropout procedure during a boosting iteration
- `xgboost_dart_mode`, default = `false`, type = bool
 - used only in `dart`
 - set this to `true`, if you want to use XGBoost DART mode
- `uniform_drop`, default = `false`, type = bool
 - used only in `dart`
 - set this to `true`, if you want to use uniform drop
- `drop_seed`, default = `4`, type = int
 - used only in `dart`
 - random seed to choose dropping models
- `top_rate`, default = `0.2`, type = double, constraints: `0.0 <= top_rate <= 1.0`
 - 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`
 - used for the categorical features
 - minimal number of data per categorical group
- **max_cat_threshold** , default = `32`, type = int, constraints: `max_cat_threshold > 0`
 - used for the categorical features
 - limit number of split points considered for categorical features. See [the documentation on how LightGBM finds optimal splits for categorical features](#) for more details
 - can be used to speed up training
- **cat_l2** , default = `10.0`, type = double, constraints: `cat_l2 >= 0.0`
 - used for the categorical features
 - L2 regularization in categorical split
- **cat_smooth** , default = `10.0`, type = double, constraints: `cat_smooth >= 0.0`
 - used for the categorical features
 - this can reduce the effect of noises in categorical features, especially for categories with few data
- **max_cat_to_onehot** , default = `4`, type = int, constraints: `max_cat_to_onehot > 0`
 - used for the categorical features
 - when number of categories of one feature smaller than or equal to `max_cat_to_onehot`, one-vs-other split algorithm will be used
- **top_k** , default = `20`, type = int, aliases: `topk`, constraints: `top_k > 0`
 - used only in [voting](#) tree learner, refer to [Voting parallel](#)
 - set this to larger value for more accurate result, but it will slow down the training speed
- **monotone_constraints** , default = `None`, type = multi-int, aliases: `mc`, `monotone_constraint`, `monotonic_cst`
 - 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 the 1st feature, non-constraint for the 2nd feature and increasing for the 3rd feature
- **monotone_constraints_method** , default = `basic`, type = enum, options: `basic`, `intermediate`, `advanced`, aliases: `monotone_constrainting_method`, `mc_method`
 - used only if `monotone_constraints` is set
 - monotone constraints method
 - * `basic`, the most basic monotone constraints method. It does not slow down the training speed at all, but over-constrains the predictions

- * `intermediate`, a more advanced method, which may slow down the training speed very slightly. However, this method is much less constraining than the basic method and should significantly improve the results
- * `advanced`, an even more advanced method, which may slow down the training speed. However, this method is even less constraining than the intermediate method and should again significantly improve the results
- `monotone_penalty`, default = `0.0`, type = double, aliases: `monotone_splits_penalty`, `ms_penalty`, `mc_penalty`, constraints: `monotone_penalty >= 0.0`
 - used only if `monotone_constraints` is set
 - `monotone penalty`: a penalization parameter X forbids any monotone splits on the first X (rounded down) level(s) of the tree. The penalty applied to monotone splits on a given depth is a continuous, increasing function the penalization parameter
 - if `0.0` (the default), no penalization is applied
- `feature_contrib`, 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_contrib[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
- `cegb_tradeoff`, default = `1.0`, type = double, constraints: `cegb_tradeoff >= 0.0`
 - cost-effective gradient boosting multiplier for all penalties
- `cegb_penalty_split`, default = `0.0`, type = double, constraints: `cegb_penalty_split >= 0.0`
 - cost-effective gradient-boosting penalty for splitting a node
- `cegb_penalty_feature_lazy`, default = `0, 0, ..., 0`, type = multi-double
 - cost-effective gradient boosting penalty for using a feature
 - applied per data point
- `cegb_penalty_feature_coupled`, default = `0, 0, ..., 0`, type = multi-double
 - cost-effective gradient boosting penalty for using a feature

- applied once per forest
- **path_smooth**, default = 0, type = double, constraints: $\text{path_smooth} \geq 0.0$
 - controls smoothing applied to tree nodes
 - helps prevent overfitting on leaves with few samples
 - if 0.0 (the default), no smoothing is applied
 - if $\text{path_smooth} > 0$ then **min_data_in_leaf** must be at least 2
 - larger values give stronger regularization
 - * the weight of each node is $w * (n / \text{path_smooth}) / (n / \text{path_smooth} + 1) + w_p / (n / \text{path_smooth} + 1)$, where n is the number of samples in the node, w is the optimal node weight to minimise the loss (approximately $-\text{sum_gradients} / \text{sum_hessians}$), and w_p is the weight of the parent node
 - * note that the parent output w_p itself has smoothing applied, unless it is the root node, so that the smoothing effect accumulates with the tree depth
- **interaction_constraints**, default = "", type = string
 - controls which features can appear in the same branch
 - by default interaction constraints are disabled, to enable them you can specify
 - * for CLI, lists separated by commas, e.g. [0, 1, 2], [2, 3]
 - * for Python-package, list of lists, e.g. [[0, 1, 2], [2, 3]]
 - * for R-package, list of character or numeric vectors, e.g. `list(c("var1", "var2", "var3"), c("var3", "var4"))` or `list(c(1L, 2L, 3L), c(3L, 4L))`. Numeric vectors should use 1-based indexing, where 1L is the first feature, 2L is the second feature, etc.
 - any two features can only appear in the same branch only if there exists a constraint containing both features
- **verbosity**, default = 1, type = int, aliases: **verbose**
 - controls the level of LightGBM's verbosity
 - < 0: Fatal, = 0: Error (Warning), = 1: Info, > 1: Debug
- **input_model**, default = "", type = string, aliases: **model_input**, **model_in**
 - filename of input model
 - for **prediction** task, this model will be applied to prediction data
 - for **train** task, training will be continued from this model
 - **Note:** can be used only in CLI version
- **output_model**, default = `LightGBM_model.txt`, type = string, aliases: **model_output**, **model_out**
 - filename of output model in training
 - **Note:** can be used only in CLI version
- **saved_feature_importance_type**, default = 0, type = int
 - the feature importance type in the saved model file
 - 0: count-based feature importance (numbers of splits are counted); 1: gain-based feature importance (values of gain are counted)
 - **Note:** can be used only in CLI version

- `snapshot_freq`, default = -1, type = int, aliases: `save_period`
 - 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
- `use_quantized_grad`, default = `false`, type = bool
 - whether to use gradient quantization when training
 - enabling this will discretize (quantize) the gradients and hessians into bins of `num_grad_quant_bins`
 - with quantized training, most arithmetics in the training process will be integer operations
 - gradient quantization can accelerate training, with little accuracy drop in most cases
 - **Note:** works only with cpu and cuda device type
 - *New in version 4.0.0*
- `num_grad_quant_bins`, default = 4, type = int
 - used only if `use_quantized_grad=true`
 - number of bins to quantization gradients and hessians
 - with more bins, the quantized training will be closer to full precision training
 - **Note:** works only with cpu and cuda device type
 - *New in version 4.0.0*
- `quant_train_renew_leaf`, default = `false`, type = bool
 - used only if `use_quantized_grad=true`
 - whether to renew the leaf values with original gradients when quantized training
 - renewing is very helpful for good quantized training accuracy for ranking objectives
 - **Note:** works only with cpu and cuda device type
 - *New in version 4.0.0*
- `stochastic_rounding`, default = `true`, type = bool
 - used only if `use_quantized_grad=true`
 - whether to use stochastic rounding in gradient quantization
 - **Note:** works only with cpu and cuda device type
 - *New in version 4.0.0*

6.4 IO Parameters

6.4.1 Dataset Parameters

- `linear_tree`, default = `false`, type = bool, aliases: `linear_trees`
 - fit piecewise linear gradient boosting tree
 - tree splits are chosen in the usual way, but the model at each leaf is linear instead of constant
 - the linear model at each leaf includes all the numerical features in that leaf's branch

- the first tree has constant leaf values
- categorical features are used for splits as normal but are not used in the linear models
- missing values should not be encoded as `0`. Use `np.nan` for Python, `NA` for the CLI, and `NA, NA_real_, or NA_integer_` for R
- it is recommended to rescale data before training so that features have similar mean and standard deviation
- **Note:** works only with `cpu`, `gpu` device type and `serial` tree learner
- **Note:** `regression_l1` objective is not supported with linear tree boosting
- **Note:** setting `linear_tree=true` significantly increases the memory use of LightGBM
- **Note:** if you specify `monotone_constraints`, constraints will be enforced when choosing the split points, but not when fitting the linear models on leaves
- `max_bin`, default = 255, type = int, aliases: `max_bins`, constraints: `max_bin > 1`
 - max number of bins that feature values will be bucketed in
 - small number of bins may reduce training accuracy but may increase general power (deal with over-fitting)
 - LightGBM will auto compress memory according to `max_bin`. For example, LightGBM will use `uint8_t` for feature value if `max_bin=255`
- `max_bin_by_feature`, default = `None`, type = multi-int
 - max number of bins for each feature
 - if not specified, will use `max_bin` for all features
- `min_data_in_bin`, default = 3, type = int, constraints: `min_data_in_bin > 0`
 - minimal number of data inside one bin
 - use this to avoid one-data-one-bin (potential over-fitting)
- `bin_construct_sample_cnt`, default = 200000, type = int, aliases: `subsample_for_bin`, constraints: `bin_construct_sample_cnt > 0`
 - number of data that sampled to construct feature discrete bins
 - setting this to larger value will give better training result, but may increase data loading time
 - set this to larger value if data is very sparse
 - **Note:** don't set this to small values, otherwise, you may encounter unexpected errors and poor accuracy
- `data_random_seed`, default = 1, type = int, aliases: `data_seed`
 - random seed for sampling data to construct histogram bins
- `is_enable_sparse`, default = `true`, type = bool, aliases: `is_sparse, enable_sparse, sparse`
 - used to enable/disable sparse optimization
- `enable_bundle`, default = `true`, type = bool, aliases: `is_enable_bundle, bundle`
 - set this to `false` to disable Exclusive Feature Bundling (EFB), which is described in [LightGBM: A Highly Efficient Gradient Boosting Decision Tree](#)
 - **Note:** disabling this may cause the slow training speed for sparse datasets
- `use_missing`, default = `true`, type = bool
 - set this to `false` to disable the special handle of missing value
- `zero_as_missing`, default = `false`, type = bool

- set this to `true` to treat all zero as missing values (including the unshown values in LibSVM / sparse matrices)
- set this to `false` to use `na` for representing missing values
- `feature_pre_filter`, default = `true`, type = `bool`
 - set this to `true` (the default) to tell LightGBM to ignore the features that are unsplittable based on `min_data_in_leaf`
 - as dataset object is initialized only once and cannot be changed after that, you may need to set this to `false` when searching parameters with `min_data_in_leaf`, otherwise features are filtered by `min_data_in_leaf` firstly if you don't reconstruct dataset object
 - **Note:** setting this to `false` may slow down the training
- `pre_partition`, default = `false`, type = `bool`, aliases: `is_pre_partition`
 - used for distributed learning (excluding the `feature_parallel` mode)
 - `true` if training data are pre-partitioned, and different machines use different partitions
- `two_round`, default = `false`, type = `bool`, aliases: `two_round_loading`, `use_two_round_loading`
 - set this to `true` if data file is too big to fit in memory
 - by default, LightGBM will map data file to memory and load features from memory. This will provide faster data loading speed, but may cause run out of memory error when the data file is very big
 - **Note:** works only in case of loading data directly from text file
- `header`, default = `false`, type = `bool`, aliases: `has_header`
 - set this to `true` if input data has header
 - **Note:** works only in case of loading data directly from text 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`
 - if omitted, the first column in the training data is used as the label
 - **Note:** works only in case of loading data directly from text 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 text 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`
 - **Note:** weights should be non-negative
- `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 text file
- **Note:** data should be grouped by `query_id`, for more information, see [Query Data](#)
- **Note:** index starts from 0 and it doesn't count the label column when passing type is `int`, e.g. when label is column_0 and `query_id` is column_1, the correct parameter is `query=0`
- `ignore_column`, default = "", type = multi-int or string, aliases: `ignore_feature`, `blacklist`
 - used to specify some ignoring columns in training
 - use number for index, e.g. `ignore_column=0,1,2` means column_0, column_1 and column_2 will be ignored
 - add a prefix `name`: for column name, e.g. `ignore_column=name:c1,c2,c3` means c1, c2 and c3 will be ignored
 - **Note:** works only in case of loading data directly from text 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`, `categorical_features`
 - 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:** all values will be cast to `int32` (integer codes will be extracted from pandas categoricals in the Python-package)
 - **Note:** index starts from 0 and it doesn't count the label column when passing type is `int`
 - **Note:** all values should be less than `Int32.MaxValue` (2147483647)
 - **Note:** using large values could be memory consuming. Tree decision rule works best when categorical features are presented by consecutive integers starting from zero
 - **Note:** all negative values will be treated as **missing values**
 - **Note:** the output cannot be monotonically constrained with respect to a categorical feature
 - **Note:** floating point numbers in categorical features will be rounded towards 0
- `forcedbins_filename`, default = "", type = string
 - path to a .json file that specifies bin upper bounds for some or all features
 - .json file should contain an array of objects, each containing the word `feature` (integer feature index) and `bin_upper_bound` (array of thresholds for binning)
 - see [this file](#) as an example
- `save_binary`, default = `false`, type = bool, aliases: `is_save_binary`, `is_save_binary_file`
 - if `true`, LightGBM will save the dataset (including validation data) to a binary file. This speed ups the data loading for the next time

- **Note:** `init_score` is not saved in binary file
- **Note:** can be used only in CLI version; for language-specific packages you can use the correspondent function
- `precise_float_parser`, default = `false`, type = `bool`
 - use precise floating point number parsing for text parser (e.g. CSV, TSV, LibSVM input)
 - **Note:** setting this to `true` may lead to much slower text parsing
- `parser_config_file`, default = "", type = `string`
 - path to a `.json` file that specifies customized parser initialized configuration
 - see [lightgbm-transform](#) for usage examples
 - **Note:** `lightgbm-transform` is not maintained by LightGBM's maintainers. Bug reports or feature requests should go to [issues page](#)
 - *New in version 4.0.0*

6.4.2 Predict Parameters

- `start_iteration_predict`, default = `0`, type = `int`
 - used only in `prediction` task
 - used to specify from which iteration to start the prediction
 - `<= 0` means from the first iteration
- `num_iteration_predict`, default = `-1`, type = `int`
 - used only in `prediction` task
 - used to specify how many trained iterations will be used in prediction
 - `<= 0` means no limit
- `predict_raw_score`, default = `false`, type = `bool`, aliases: `is_predict_raw_score`, `predict_rawscore`, `raw_score`
 - used only in `prediction` task
 - set this to `true` to predict only the raw scores
 - set this to `false` to predict transformed scores
- `predict_leaf_index`, default = `false`, type = `bool`, aliases: `is_predict_leaf_index`, `leaf_index`
 - used only in `prediction` task
 - set this to `true` to predict with leaf index of all trees
- `predict_contrib`, default = `false`, type = `bool`, aliases: `is_predict_contrib`, `contrib`
 - used only in `prediction` task
 - set this to `true` to estimate [SHAP values](#), which represent how each feature contributes to each prediction
 - produces `#features + 1` values where the last value is the expected value of the model output over the training data
 - **Note:** if you want to get more explanation for your model's predictions using SHAP values like SHAP interaction values, you can install [shap package](#)
 - **Note:** unlike the `shap` package, with `predict_contrib` we return a matrix with an extra column, where the last column is the expected value

- **Note:** this feature is not implemented for linear trees
- `predict_disable_shape_check`, default = `false`, type = `bool`
 - used only in `prediction` task
 - control whether or not LightGBM raises an error when you try to predict on data with a different number of features than the training data
 - if `false` (the default), a fatal error will be raised if the number of features in the dataset you predict on differs from the number seen during training
 - if `true`, LightGBM will attempt to predict on whatever data you provide. This is dangerous because you might get incorrect predictions, but you could use it in situations where it is difficult or expensive to generate some features and you are very confident that they were never chosen for splits in the model
 - **Note:** be very careful setting this parameter to `true`
- `pred_early_stop`, default = `false`, type = `bool`
 - used only in `prediction` task
 - used only in `classification` and `ranking` applications
 - used only for predicting normal or raw scores
 - if `true`, will use early-stopping to speed up the prediction. May affect the accuracy
 - **Note:** cannot be used with `rf` boosting type or custom objective function
- `pred_early_stop_freq`, default = `10`, type = `int`
 - used only in `prediction` task and if `pred_early_stop=true`
 - the frequency of checking early-stopping prediction
- `pred_early_stop_margin`, default = `10.0`, type = `double`
 - used only in `prediction` task and if `pred_early_stop=true`
 - the threshold of margin in early-stopping prediction
- `output_result`, default = `LightGBM_predict_result.txt`, type = `string`, aliases: `predict_result`, `prediction_result`, `predict_name`, `prediction_name`, `pred_name`, `name_pred`
 - used only in `prediction` task
 - filename of prediction result
 - **Note:** can be used only in CLI version

6.4.3 Convert Parameters

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

- **Note:** can be used only in CLI version

6.5 Objective Parameters

- **objective_seed**, default = 5, type = int
 - used only in `rank_xendcg` objective
 - random seed for objectives, if random process is needed
- **num_class**, default = 1, type = int, aliases: `num_classes`, constraints: `num_class > 0`
 - used only in `multi-class` classification application
- **is_unbalance**, default = `false`, type = bool, aliases: `unbalance`, `unbalanced_sets`
 - used only in `binary` and `multiclassova` applications
 - set this to `true` if training data are unbalanced
 - **Note:** while enabling this should increase the overall performance metric of your model, it will also result in poor estimates of the individual class probabilities
 - **Note:** this parameter cannot be used at the same time with `scale_pos_weight`, choose only **one** of them
- **scale_pos_weight**, default = `1.0`, type = double, constraints: `scale_pos_weight > 0.0`
 - used only in `binary` and `multiclassova` applications
 - weight of labels with positive class
 - **Note:** while enabling this should increase the overall performance metric of your model, it will also result in poor estimates of the individual class probabilities
 - **Note:** this parameter cannot be used at the same time with `is_unbalance`, choose only **one** of them
- **sigmoid**, default = `1.0`, type = double, constraints: `sigmoid > 0.0`
 - used only in `binary` and `multiclassova` classification and in `lambdarank` applications
 - parameter for the sigmoid function
- **boost_from_average**, default = `true`, type = bool
 - used only in `regression`, `binary`, `multiclassova` and `cross-entropy` applications
 - adjusts initial score to the mean of labels for faster convergence
- **reg_sqrt**, default = `false`, type = bool
 - used only in `regression` application
 - used to fit `sqrt(label)` instead of original values and prediction result will be also automatically converted to `prediction^2`
 - might be useful in case of large-range labels
- **alpha**, default = `0.9`, type = double, constraints: `alpha > 0.0`
 - used only in `huber` and `quantile` regression applications
 - parameter for `Huber` loss and `Quantile` regression
- **fair_c**, default = `1.0`, type = double, constraints: `fair_c > 0.0`
 - used only in `fair` regression application
 - parameter for `Fair` loss

- `poisson_max_delta_step`, default = `0.7`, type = double, constraints: `poisson_max_delta_step > 0.0`
 - used only in `poisson` regression application
 - parameter for [Poisson regression](#) to safeguard optimization
- `tweedie_variance_power`, default = `1.5`, type = double, constraints: `1.0 <= tweedie_variance_power < 2.0`
 - used only in `tweedie` regression application
 - 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
- `lambdarank_truncation_level`, default = `30`, type = int, constraints: `lambdarank_truncation_level > 0`
 - used only in `lambdarank` application
 - controls the number of top-results to focus on during training, refer to “truncation level” in the Sec. 3 of [LambdaMART paper](#)
 - this parameter is closely related to the desirable cutoff k in the metric **NDCG@k** that we aim at optimizing the ranker for. The optimal setting for this parameter is likely to be slightly higher than k (e.g., $k + 3$) to include more pairs of documents to train on, but perhaps not too high to avoid deviating too much from the desired target metric **NDCG@k**
- `lambdarank_norm`, default = `true`, type = bool
 - used only in `lambdarank` application
 - set this to `true` to normalize the lambdas for different queries, and improve the performance for unbalanced data
 - set this to `false` to enforce the original lambdarank algorithm
- `label_gain`, default = `0, 1, 3, 7, 15, 31, 63, ..., 2^30-1`, type = multi-double
 - used only in `lambdarank` application
 - relevant gain for labels. For example, the gain of label 2 is 3 in case of default label gains
 - separate by `,`
- `lambdarank_position_bias_regularization`, default = `0.0`, type = double, constraints: `lambdarank_position_bias_regularization >= 0.0`
 - used only in `lambdarank` application when positional information is provided and position bias is modeled
 - larger values reduce the inferred position bias factors
 - *New in version 4.1.0*

6.6 Metric Parameters

- `metric`, default = `""`, type = multi-enum, aliases: `metrics`, `metric_types`
 - metric(s) to be evaluated on the evaluation set(s)
 - * `""` (empty string or not specified) means that metric corresponding to specified `objective` will be used (this is possible only for pre-defined objective functions, otherwise no evaluation metric will be added)

- * "None" (string, **not** a None value) means that no metric will be registered, aliases: `na`, `null`, `custom`
- * `l1`, absolute loss, aliases: `mean_absolute_error`, `mae`, `regression_l1`
- * `l2`, square loss, aliases: `mean_squared_error`, `mse`, `regression_l2`, `regression`
- * `rmse`, root square loss, aliases: `root_mean_squared_error`, `l2_root`
- * `quantile`, `Quantile regression`
- * `mape`, `MAPE` loss, aliases: `mean_absolute_percentage_error`
- * `huber`, Huber loss
- * `fair`, `Fair` loss
- * `poisson`, negative log-likelihood for `Poisson` regression
- * `gamma`, negative log-likelihood for `Gamma` regression
- * `gamma_deviance`, residual deviance for `Gamma` regression
- * `tweedie`, negative log-likelihood for `Tweedie` regression
- * `ndcg`, `NDCG`, aliases: `lambdarank`, `rank_xendcg`, `xendcg`, `xe_ndcg`, `xe_ndcg_mart`, `xendcg_mart`
- * `map`, `MAP`, aliases: `mean_average_precision`
- * `auc`, `AUC`
- * `average_precision`, average precision score
- * `r2`, `R-squared`
- * `binary_logloss`, `log loss`, aliases: `binary`
- * `binary_error`, for one sample: 0 for correct classification, 1 for error classification
- * `auc_mu`, `AUC-mu`
- * `multi_logloss`, log loss for multi-class classification, aliases: `multiclass`, `softmax`, `multiclassova`, `multiclass_ova`, `ova`, `ovr`
- * `multi_error`, error rate for multi-class classification
- * `cross_entropy`, cross-entropy (with optional linear weights), aliases: `xentropy`
- * `cross_entropy_lambda`, “intensity-weighted” cross-entropy, aliases: `xentlambda`
- * `kullback_leibler`, `Kullback-Leibler` divergence, aliases: `kldiv`
- support multiple metrics, separated by ,
- `metric_freq`, default = 1, type = int, aliases: `output_freq`, constraints: `metric_freq > 0`
 - frequency for metric output
 - **Note:** can be used only in CLI version
- `is_provide_training_metric`, default = `false`, type = `bool`, aliases: `training_metric`, `is_training_metric`, `train_metric`
 - set this to `true` to output metric result over training dataset
 - **Note:** can be used only in CLI version
- `eval_at`, default = 1,2,3,4,5, type = multi-int, aliases: `ndcg_eval_at`, `ndcg_at`, `map_eval_at`, `map_at`
 - used only with `ndcg` and `map` metrics

- NDCG and MAP evaluation positions, separated by ,
- `multi_error_top_k`, default = 1, type = int, constraints: `multi_error_top_k > 0`
 - used only with `multi_error` metric
 - threshold for top-k multi-error metric
 - the error on each sample is 0 if the true class is among the top `multi_error_top_k` predictions, and 1 otherwise
 - * more precisely, the error on a sample is 0 if there are at least `num_classes - multi_error_top_k` predictions strictly less than the prediction on the true class
 - when `multi_error_top_k=1` this is equivalent to the usual multi-error metric
- `auc_mu_weights`, default = `None`, type = multi-double
 - used only with `auc_mu` metric
 - list representing flattened matrix (in row-major order) giving loss weights for classification errors
 - list should have `n * n` elements, where `n` is the number of classes
 - the matrix co-ordinate `[i, j]` should correspond to the `i * n + j`-th element of the list
 - if not specified, will use equal weights for all classes

6.7 Network Parameters

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

6.8 GPU Parameters

- `gpu_platform_id`, default = -1, type = int
 - used only with gpu device type
 - 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 or CUDA device ID. Each GPU in the selected platform has a unique device ID
 - -1 means the default device in the selected platform
 - **Note:** refer to [GPU Targets](#) for more details
- `gpu_use_dp`, default = `false`, type = bool
 - set this to `true` to use double precision math on GPU (by default single precision is used)
 - **Note:** can be used only in OpenCL implementation (`device_type="gpu"`), in CUDA implementation only double precision is currently supported
- `num_gpu`, default = 1, type = int, constraints: `num_gpu > 0`
 - number of GPUs
 - **Note:** can be used only in CUDA implementation (`device_type="cuda"`)

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.

If the name of data file is `train.txt`, the initial score file should be named as `train.txt.init` and placed in the same folder as the data file. In this case, LightGBM will auto load initial score file if it exists.

If binary data files exist for raw data file `train.txt`, for example in the name `train.txt.bin`, then the initial score file should be named as `train.txt.bin.init`.

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. Weights should be non-negative.

The weight file corresponds with data file line by line, and has per weight per line.

And if the name of data file is `train.txt`, the weight file should be named as `train.txt.weight` and placed in the same folder as the data file. In this case, LightGBM will load the weight file automatically if it exists.

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

6.9.3 Query Data

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

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

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

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

```
[27, 18, 67, ...]
```

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

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.

PARAMETERS TUNING

This page contains parameters tuning guides for different scenarios.

List of other helpful links

- [Parameters](#)
- [Python API](#)
- [FLAML](#) for automated hyperparameter tuning
- [Optuna](#) for automated hyperparameter tuning

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

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

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

1. `num_leaves`. This is the main parameter to control the complexity of the tree model. Theoretically, we can set `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. 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. If you set `max_depth`, also explicitly set `num_leaves` to some value $\leq 2^{\max_depth}$.

7.2 For Faster Speed

7.2.1 Add More Computational Resources

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

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

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

7.2.2 Use a GPU-enabled version of LightGBM

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

7.2.3 Grow Shallower Trees

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

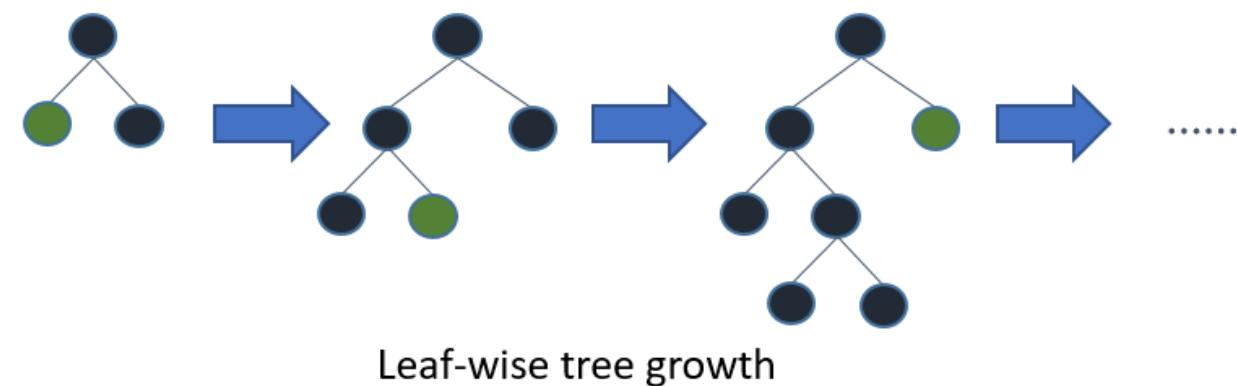
The suggestions below will speed up training, but might hurt training accuracy.

Decrease `max_depth`

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

Decrease `num_leaves`

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



Because of this growth strategy, it isn't straightforward to use `max_depth` alone to limit the complexity of trees. The `num_leaves` parameter sets the maximum number of nodes per tree. Decrease `num_leaves` to reduce training time.

Increase `min_gain_to_split`

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

Increase `min_data_in_leaf` and `min_sum_hessian_in_leaf`

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

This can be prevented indirectly with parameters like `max_depth` and `num_leaves`, but LightGBM also offers parameters to help you directly avoid adding these overly-specific tree nodes.

- `min_data_in_leaf`: Minimum number of observations that must fall into a tree node for it to be added.
- `min_sum_hessian_in_leaf`: Minimum sum of the Hessian (second derivative of the objective function evaluated for each observation) for observations in a leaf. For some regression objectives, this is just the minimum number of records that have to fall into each node. For classification objectives, it represents a sum over a distribution of probabilities. See [this Stack Overflow answer](#) for a good description of how to reason about values of this parameter.

7.2.4 Grow Less Trees

Decrease `num_iterations`

The `num_iterations` parameter controls the number of boosting rounds that will be performed. Since LightGBM uses decision trees as the learners, this can also be thought of as “number of trees”.

If you try changing `num_iterations`, change the `learning_rate` as well. `learning_rate` will not have any impact on training time, but it will impact the training accuracy. As a general rule, if you reduce `num_iterations`, you should increase `learning_rate`.

Choosing the right value of `num_iterations` and `learning_rate` is highly dependent on the data and objective, so these parameters are often chosen from a set of possible values through hyperparameter tuning.

Decrease `num_iterations` to reduce training time.

Use Early Stopping

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

That “number of consecutive rounds” is controlled by the parameter `early_stopping_round`. For example, `early_stopping_round=1` says “the first time accuracy on the validation set does not improve, stop training”.

Set `early_stopping_round` and provide a validation set to possibly reduce training time.

7.2.5 Consider Fewer Splits

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

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

Enable Feature Pre-Filtering When Creating Dataset

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

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

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

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

Decrease `max_bin` or `max_bin_by_feature` When Creating Dataset

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

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

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

Increase `min_data_in_bin` When Creating Dataset

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

Increase `min_data_in_bin` to reduce training time.

Decrease `feature_fraction`

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

Decrease `feature_fraction` to reduce training time.

Decrease `max_cat_threshold`

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

Decrease `max_cat_threshold` to reduce training time.

7.2.6 Use Less Data

Use Bagging

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

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

Decrease `bagging_fraction` to reduce training time.

7.2.7 Save Constructed Datasets with `save_binary`

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

7.3 For Better Accuracy

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

7.4 Deal with Over-fitting

- Use small `max_bin`
- Use small `num_leaves`
- Use `min_data_in_leaf` and `min_sum_hessian_in_leaf`
- Use bagging by set `bagging_fraction` and `bagging_freq`
- Use feature sub-sampling by set `feature_fraction`
- Use bigger training data
- Try `lambda_11`, `lambda_12` and `min_gain_to_split` for regularization
- Try `max_depth` to avoid growing deep tree
- Try `extra_trees`
- Try increasing `path_smooth`

Copyright

Copyright (c) 2016 Microsoft Corporation. All rights reserved. Licensed under the MIT License. See LICENSE file in the project root for license information.

Note

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

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

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

Defines**C_API_DTYPE_FLOAT32 (0)**

float32 (single precision float).

C_API_DTYPE_FLOAT64 (1)

float64 (double precision float).

C_API_DTYPE_INT32 (2)

int32.

C_API_DTYPE_INT64 (3)

int64.

C_API_FEATURE_IMPORTANCE_GAIN (1)

Gain type of feature importance.

C_API_FEATURE_IMPORTANCE_SPLIT (0)

Split type of feature importance.

C_API_MATRIX_TYPE_CSC (1)

CSC sparse matrix type.

C_API_MATRIX_TYPE_CSR (0)

CSR sparse matrix type.

C_API_PREDICT CONTRIB (3)

Predict feature contributions (SHAP values).

C_API_PREDICT LEAF INDEX (2)

Predict leaf index.

C_API_PREDICT NORMAL (0)

Normal prediction, with transform (if needed).

C_API_PREDICT RAW SCORE (1)

Predict raw score.

INLINE_FUNCTION inline

Inline specifier.

THREAD_LOCAL thread_local

Thread local specifier.

Typedefs

typedef void *BoosterHandle

Handle of booster.

typedef void *ByteBufferHandle

Handle of ByteBuffer.

typedef void *DatasetHandle

Handle of dataset.

typedef void *FastConfigHandle

Handle of FastConfig.

Functions

static char *LastErrorMsg()

Handle of error message.

Returns

Error message

LIGHTGBM_C_EXPORT int **LGBM_BoosterAddValidData**(*BoosterHandle* handle, const *DatasetHandle* valid_data)

Add new validation data to booster.

Parameters

- **handle** – Handle of booster
- **valid_data** – Validation dataset

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterCalcNumPredict**(*BoosterHandle* handle, int num_row, int predict_type, int start_iteration, int num_iteration, int64_t *out_len)

Get number of predictions.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Create a new boosting learner.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Load an existing booster from model file.

Parameters

- **filename** – Filename of model

- **out_num_iterations** – [out] Number of iterations of this booster
- **out** – [out] Handle of created booster

Returns

0 when succeed, -1 when failure happens

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

Dump model to JSON.

Parameters

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

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterFeatureImportance**(*BoosterHandle* handle, int num_iteration, int importance_type, double *out_results)

Get model feature importance.

Parameters

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

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterFree**(*BoosterHandle* handle)

Free space for booster.

Parameters

- **handle** – Handle of booster to be freed

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterFreePredictSparse**(void *indptr, int32_t *indices, void *data, int indptr_type, int data_type)

Method corresponding to `LGBM_BoosterPredictSparseOutput` to free the allocated data.

Parameters

- **indptr** – Pointer to output row headers or column headers to be deallocated
- **indices** – Pointer to sparse indices to be deallocated
- **data** – Pointer to sparse data space to be deallocated
- **indptr_type** – Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- **data_type** – Type of `data` pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterGetCurrentIteration**(*BoosterHandle* handle, int *out_iteration)

Get index of the current boosting iteration.

Parameters

- **handle** – Handle of booster
- **out_iteration** – [out] Index of the current boosting iteration

Returns

0 when succeed, -1 when failure happens

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

Get evaluation for training data and validation data.

Note

- a. You should call `LGBM_BoosterGetEvalNames` first to get the names of evaluation metrics.
- b. You should pre-allocate memory for `out_results`, you can get its length by `LGBM_BoosterGetEvalCounts`.

Parameters

- **handle** – Handle of booster
- **data_idx** – Index of data, 0: training data, 1: 1st validation data, 2: 2nd validation data and so on
- **out_len** – [out] Length of output result
- **out_results** – [out] Array with evaluation results

Returns

0 when succeed, -1 when failure happens

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

Get number of evaluation metrics.

Parameters

- **handle** – Handle of booster
- **out_len** – [out] Total number of evaluation metrics

Returns

0 when succeed, -1 when failure happens

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

Get names of evaluation metrics.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Get names of features.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Get leaf value.

Parameters

- **handle** – Handle of booster
- **tree_idx** – Index of tree

- **leaf_idx** – Index of leaf
- **out_val** – [out] Output result from the specified leaf

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterGetLinear**(*BoosterHandle* handle, int *out)

Get int representing whether booster is fitting linear trees.

Parameters

- **handle** – Handle of booster
- **out** – [out] The address to hold linear trees indicator

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterGetLoadedParam**(*BoosterHandle* handle, int64_t buffer_len, int64_t *out_len, char *out_str)

Get parameters as JSON string.

Parameters

- **handle** – Handle of booster
- **buffer_len** – Allocated space for string
- **out_len** – [out] Actual size of string
- **out_str** – [out] JSON string containing parameters

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterGetLowerBoundValue**(*BoosterHandle* handle, double *out_results)

Get model lower bound value.

Parameters

- **handle** – Handle of booster
- **out_results** – [out] Result pointing to min value

Returns

0 when succeed, -1 when failure happens

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

Get number of classes.

Parameters

- **handle** – Handle of booster
- **out_len** – [out] Number of classes

Returns

0 when succeed, -1 when failure happens

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

Get number of features.

Parameters

- **handle** – Handle of booster

- **out_len** – [out] Total number of features

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterGetNumPredict**(*BoosterHandle* handle, int data_idx, int64_t *out_len)

Get number of predictions for training data and validation data (this can be used to support customized evaluation functions).

Parameters

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

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterGetPredict**(*BoosterHandle* handle, int data_idx, int64_t *out_len, double *out_result)

Get prediction for training data and validation data.

Note

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

Parameters

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

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterGetUpperBoundValue**(*BoosterHandle* handle, double *out_results)

Get model upper bound value.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Load an existing booster from string.

Parameters

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

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterMerge**(*BoosterHandle* handle, *BoosterHandle* other_handle)

Merge model from other_handle into handle.

Parameters

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

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterNumberOfTotalModel**(*BoosterHandle* handle, int *out_models)

Get number of weak sub-models.

Parameters

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

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterNumModelPerIteration**(*BoosterHandle* handle, int *out_tree_per_iteration)

Get number of trees per iteration.

Parameters

- **handle** – Handle of booster
- **out_tree_per_iteration** – [out] Number of trees per iteration

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterPredictForArrow**(*BoosterHandle* handle, int64_t n_chunks, const struct ArrowArray *chunks, const struct ArrowSchema *schema, int predict_type, int start_iteration, int num_iteration, const char *parameter, int64_t *out_len, double *out_result)

Make prediction for a new dataset.

Note

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

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

Parameters

- **handle** – Handle of booster
- **n_chunks** – The number of Arrow arrays passed to this function
- **chunks** – Pointer to the list of Arrow arrays
- **schema** – Pointer to the schema of all Arrow arrays
- **predict_type** – What should be predicted
 - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
 - C_API_PREDICT_RAW_SCORE: raw score;
 - C_API_PREDICT_LEAF_INDEX: leaf index;
 - C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
- **start_iteration** – Start index of the iteration to predict
- **num_iteration** – Number of iteration for prediction, <= 0 means no limit
- **parameter** – Other parameters for prediction, e.g. early stopping for prediction
- **out_len** – [out] Length of output result
- **out_result** – [out] Pointer to array with predictions

Returns

0 when succeed, -1 when failure happens

```
LIGHTGBM_C_EXPORT int LGBM_BoosterPredictForCSC(BoosterHandle handle, const void *col_ptr, int
                                                col_ptr_type, const int32_t *indices, const void
                                                *data, int data_type, int64_t ncol_ptr, int64_t nelem,
                                                int64_t num_row, int predict_type, int
                                                start_iteration, int num_iteration, const char
                                                *parameter, int64_t *out_len, double *out_result)
```

Make prediction for a new dataset in CSC format.

Note

You should pre-allocate memory for `out_result`:

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

Parameters

- **handle** – Handle of booster
- **col_ptr** – Pointer to column headers
- **col_ptr_type** – Type of `col_ptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- **indices** – Pointer to row indices
- **data** – Pointer to the data space

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

Returns

0 when succeed, -1 when failure happens

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

Make prediction for a new dataset in CSR format.

Note

You should pre-allocate memory for `out_result`:

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

Parameters

- **handle** – Handle of booster
- **indptr** – Pointer to row headers
- **indptr_type** – Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- **indices** – Pointer to column indices
- **data** – Pointer to the data space
- **data_type** – Type of data pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`

- **nindptr** – Number of rows in the matrix + 1
- **nelem** – Number of nonzero elements in the matrix
- **num_col** – Number of columns
- **predict_type** – What should be predicted
 - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
 - C_API_PREDICT_RAW_SCORE: raw score;
 - C_API_PREDICT_LEAF_INDEX: leaf index;
 - C_API_PREDICT_CONTRIB: feature contributions (SHAP values)
- **start_iteration** – Start index of the iteration to predict
- **num_iteration** – Number of iterations for prediction, <= 0 means no limit
- **parameter** – Other parameters for prediction, e.g. early stopping for prediction
- **out_len** – [out] Length of output result
- **out_result** – [out] Pointer to array with predictions

Returns

0 when succeed, -1 when failure happens

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

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

Note

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

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

Parameters

- **handle** – Handle of booster
- **indptr** – Pointer to row headers
- **indptr_type** – Type of **indptr**, can be C_API_DTYPE_INT32 or C_API_DTYPE_INT64
- **indices** – Pointer to column indices
- **data** – Pointer to the data space

- **data_type** – Type of data pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`
- **nindptr** – Number of rows in the matrix + 1
- **nelem** – Number of nonzero elements in the matrix
- **num_col** – Number of columns
- **predict_type** – What should be predicted
 - `C_API_PREDICT_NORMAL`: normal prediction, with transform (if needed);
 - `C_API_PREDICT_RAW_SCORE`: raw score;
 - `C_API_PREDICT_LEAF_INDEX`: leaf index;
 - `C_API_PREDICT_CONTRIB`: feature contributions (SHAP values)
- **start_iteration** – Start index of the iteration to predict
- **num_iteration** – Number of iterations for prediction, ≤ 0 means no limit
- **parameter** – Other parameters for prediction, e.g. early stopping for prediction
- **out_len** – [out] Length of output result
- **out_result** – [out] Pointer to array with predictions

Returns

0 when succeed, -1 when failure happens

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

Faster variant of `LGBM_BoosterPredictForCSRSingleRow`.

Score single rows after setup with `LGBM_BoosterPredictForCSRSingleRowFastInit`.

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

Note

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

Note

You should pre-allocate memory for `out_result`:

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

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

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

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

Parameters

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

Returns

0 when it succeeds, -1 when failure happens

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

Make prediction for file.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Make prediction for a new dataset.

 **Note**

You should pre-allocate memory for `out_result`:

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

Parameters

- **handle** – Handle of booster
- **data** – Pointer to the data space
- **data_type** – Type of data pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`

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

Returns

0 when succeed, -1 when failure happens

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

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

Note

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

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

Parameters

- **handle** – Handle of booster
- **data** – Pointer to the data space
- **data_type** – Type of data pointer, can be **C_API_DTYPE_FLOAT32** or **C_API_DTYPE_FLOAT64**
- **nrow** – Number of rows
- **ncol** – Number columns
- **predict_type** – What should be predicted
 - C_API_PREDICT_NORMAL: normal prediction, with transform (if needed);
 - C_API_PREDICT_RAW_SCORE: raw score;

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

Returns

0 when succeed, -1 when failure happens

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

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

Note

You should pre-allocate memory for `out_result`:

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

Parameters

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

- **out_len** – **[out]** Length of output result
- **out_result** – **[out]** Pointer to array with predictions

Returns

0 when succeed, -1 when failure happens

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

Faster variant of `LGBM_BoosterPredictForMatSingleRow`.

Score a single row after setup with `LGBM_BoosterPredictForMatSingleRowFastInit`.

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

Note

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

Parameters

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

Returns

0 when it succeeds, -1 when failure happens

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

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

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

Parameters

- **handle** – Booster handle
- **predict_type** – What should be predicted
 - `C_API_PREDICT_NORMAL`: normal prediction, with transform (if needed);
 - `C_API_PREDICT_RAW_SCORE`: raw score;

- `C_API_PREDICT_LEAF_INDEX`: leaf index;
- `C_API_PREDICT CONTRIB`: feature contributions (SHAP values)
- `start_iteration` – Start index of the iteration to predict
- `num_iteration` – Number of iterations for prediction, $<= 0$ means no limit
- `data_type` – Type of `data` pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`
- `ncol` – Number of columns
- `parameter` – Other parameters for prediction, e.g. early stopping for prediction
- `out_fastConfig` – `[out]` `FastConfig` object with which you can call `LGBM_BoosterPredictForMatSingleRowFast`

Returns

0 when it succeeds, -1 when failure happens

`LIGHTGBM_C_EXPORT int LGBM_BoosterPredictSparseOutput(BoosterHandle handle, const void *indptr, int indptr_type, const int32_t *indices, const void *data, int data_type, int64_t nindptr, int64_t nelem, int64_t num_col_or_row, int predict_type, int start_iteration, int num_iteration, const char *parameter, int matrix_type, int64_t *out_len, void **out_indptr, int32_t **out_indices, void **out_data)`

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

Note

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

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

Parameters

- `handle` – Handle of booster
- `indptr` – Pointer to row headers for CSR or column headers for CSC
- `indptr_type` – Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- `indices` – Pointer to column indices for CSR or row indices for CSC
- `data` – Pointer to the data space
- `data_type` – Type of `data` pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`
- `nindptr` – Number of entries in `indptr`
- `nelem` – Number of nonzero elements in the matrix
- `num_col_or_row` – Number of columns for CSR or number of rows for CSC
- `predict_type` – What should be predicted, only feature contributions supported currently

- `C_API_PREDICT CONTRIB`: feature contributions (SHAP values)
- **start_iteration** – Start index of the iteration to predict
- **num_iteration** – Number of iterations for prediction, $<= 0$ means no limit
- **parameter** – Other parameters for prediction, e.g. early stopping for prediction
- **matrix_type** – Type of matrix input and output, can be `C_API_MATRIX_TYPE_CSR` or `C_API_MATRIX_TYPE_CSC`
- **out_len** – **[out]** Length of output data and output indptr (pointer to an array with two entries where to write them)
- **out_indptr** – **[out]** Pointer to output row headers for CSR or column headers for CSC
- **out_indices** – **[out]** Pointer to sparse column indices for CSR or row indices for CSC
- **out_data** – **[out]** Pointer to sparse data space

Returns

0 when succeed, -1 when failure happens

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

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

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Reset config for booster.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Reset training data for booster.

Parameters

- **handle** – Handle of booster
- **train_data** – Training dataset

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterRollbackOneIter**(*BoosterHandle* handle)

Rollback one iteration.

Parameters

- **handle** – Handle of booster

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterSaveModel**(*BoosterHandle* handle, int start_iteration, int num_iteration, int feature_importance_type, const char *filename)

Save model into file.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Save model to string.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Set leaf value.

Parameters

- **handle** – Handle of booster

- **tree_idx** – Index of tree
- **leaf_idx** – Index of leaf
- **val** – Leaf value

Returns

0 when succeed, -1 when failure happens

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

Shuffle models.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Update the model for one iteration.

Parameters

- **handle** – Handle of booster
- **produced_empty_tree** – [out] 1 means the tree(s) produced by this iteration did not have any splits. This usually means that training is “finished” (calling this function again will not change the model’s predictions). However, that is not always the case. For example, if you have added any randomness (like column sampling by setting **feature_fraction_bynode** < 1.0), it is possible that another call to this function would produce a non-empty tree.

Returns

0 when succeed, -1 when failure happens

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

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

Note

The length of the arrays referenced by **grad** and **hess** must be equal to **num_class * num_train_data**, this is not verified by the library, the caller must ensure this.

Parameters

- **handle** – Handle of booster
- **grad** – The first order derivative (gradient) statistics
- **hess** – The second order derivative (Hessian) statistics
- **produced_empty_tree** – [out] 1 means the tree(s) produced by this iteration did not have any splits. This usually means that training is “finished” (calling this function again will not change the model’s predictions). However, that is not always the case. For example, if you

have added any randomness (like column sampling by setting `feature_fraction_bynode < 1.0`), it is possible that another call to this function would produce a non-empty tree.

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_BoosterValidateFeatureNames**(*BoosterHandle* handle, const char ***data_names*, int *data_num_features*)

Check that the feature names of the data match the ones used to train the booster.

Parameters

- **handle** – Handle of booster
- **data_names** – Array with the feature names in the data
- **data_num_features** – Number of features in the data

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_ByteBufferFree**(*ByteBufferHandle* handle)

Free space for byte buffer.

Parameters

- **handle** – Handle of byte buffer to be freed

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_ByteBufferGetAt**(*ByteBufferHandle* handle, int32_t index, uint8_t **out_val*)

Get a ByteBuffer value at an index.

Parameters

- **handle** – Handle of byte buffer to be read
- **index** – Index of value to return
- **out_val** – [out] Byte value at index to return

Returns

0 when succeed, -1 when failure happens

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

Add features from source to target.

Parameters

- **target** – The handle of the dataset to add features to
- **source** – The handle of the dataset to take features from

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetCreateByReference**(const *DatasetHandle* reference, int64_t num_total_row, *DatasetHandle* **out*)

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

Parameters

- **reference** – Used to align bin mapper with other dataset

- **num_total_row** – Number of total rows
- **out** – [out] Created dataset

Returns

0 when succeed, -1 when failure happens

```
LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromArrow(int64_t n_chunks, const struct ArrowArray
*chunks, const struct ArrowSchema *schema,
const char *parameters, const DatasetHandle
reference, DatasetHandle *out)
```

Create dataset from Arrow.

Parameters

- **n_chunks** – The number of Arrow arrays passed to this function
- **chunks** – Pointer to the list of Arrow arrays
- **schema** – Pointer to the schema of all Arrow arrays
- **parameters** – Additional parameters
- **reference** – Used to align bin mapper with other dataset, nullptr means isn't used
- **out** – [out] Created dataset

Returns

0 when succeed, -1 when failure happens

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

Create a dataset from CSC format.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Create a dataset from CSR format.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Create a dataset from CSR format through callbacks.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

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

Parameters

- **filename** – The name of the file
- **parameters** – Additional parameters
- **reference** – Used to align bin mapper with other dataset, nullptr means isn't used
- **out** – [out] A loaded dataset

Returns

0 when succeed, -1 when failure happens

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

Create dataset from dense matrix.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Create dataset from array of dense matrices.

Parameters

- **nmat** – Number of dense matrices
- **data** – Pointer to the data space
- **data_type** – Type of data pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`
- **nrow** – Number of rows
- **ncol** – Number of columns
- **is_row_major** – Pointer to the data layouts. 1 for row-major, 0 for column-major
- **parameters** – Additional parameters
- **reference** – Used to align bin mapper with other dataset, nullptr means isn't used
- **out** – [out] Created dataset

Returns

0 when succeed, -1 when failure happens

```
LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromSampledColumn(double **sample_data, int
                                                       **sample_indices, int32_t ncol, const
                                                       int *num_per_col, int32_t
                                                       num_sample_row, int32_t
                                                       num_local_row, int64_t num_dist_row,
                                                       const char *parameters,
                                                       DatasetHandle *out)
```

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

Parameters

- **sample_data** – Sampled data, grouped by the column
- **sample_indices** – Indices of sampled data
- **ncol** – Number of columns
- **num_per_col** – Size of each sampling column
- **num_sample_row** – Number of sampled rows
- **num_local_row** – Total number of rows local to machine
- **num_dist_row** – Number of total distributed rows
- **parameters** – Additional parameters
- **out** – [out] Created dataset

Returns

0 when succeed, -1 when failure happens

```
LIGHTGBM_C_EXPORT int LGBM_DatasetCreateFromSerializedReference(const void *ref_buffer, int32_t
                                                               ref_buffer_size, int64_t
                                                               num_row, int32_t
                                                               num_classes, const char
                                                               *parameters, DatasetHandle
                                                               *out)
```

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

Parameters

- **ref_buffer** – A binary representation of the dataset schema (feature groups, bins, etc.)
- **ref_buffer_size** – The size of the reference array in bytes
- **num_row** – Number of total rows the dataset will contain
- **num_classes** – Number of classes (will be used only in case of multiclass and specifying initial scores)
- **parameters** – Additional parameters
- **out** – [out] Created dataset

Returns

0 when succeed, -1 when failure happens

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

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

Parameters

- **handle** – Handle of dataset
- **filename** – The name of the file

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetFree**(*DatasetHandle* handle)

Free space for dataset.

Parameters

- **handle** – Handle of dataset to be freed

Returns

0 when succeed, -1 when failure happens

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

Get feature names of dataset.

Parameters

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

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetGetFeatureNumBin**(*DatasetHandle* handle, int feature, int *out)

Get number of bins for feature.

Parameters

- **handle** – Handle of dataset
- **feature** – Index of the feature
- **out** – **[out]** The address to hold number of bins

Returns

0 when succeed, -1 when failure happens

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

Get info vector from dataset.

Parameters

- **handle** – Handle of dataset
- **field_name** – Field name
- **out_len** – **[out]** Used to set result length

- **out_ptr** – **[out]** Pointer to the result
- **out_type** – **[out]** Type of result pointer, can be C_API_DTYPE_INT32, C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64

Returns

0 when succeed, -1 when failure happens

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

Get number of data points.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Get number of features.

Parameters

- **handle** – Handle of dataset
- **out** – **[out]** The address to hold number of features

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetGetSubset**(const *DatasetHandle* handle, const int32_t *used_row_indices, int32_t num_used_row_indices, const char *parameters, *DatasetHandle* *out)

Create subset of a data.

Parameters

- **handle** – Handle of full dataset
- **used_row_indices** – Indices used in subset
- **num_used_row_indices** – Length of used_row_indices
- **parameters** – Additional parameters
- **out** – **[out]** Subset of data

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetInitStreaming**(*DatasetHandle* dataset, int32_t has_weights, int32_t has_init_scores, int32_t has_queries, int32_t nclasses, int32_t nthreads, int32_t omp_max_threads)

Initialize the Dataset for streaming.

Parameters

- **dataset** – Handle of dataset
- **has_weights** – Whether the dataset has Metadata weights
- **has_init_scores** – Whether the dataset has Metadata initial scores

- **has_queries** – Whether the dataset has Metadata queries/groups
- **nclasses** – Number of initial score classes
- **nthreads** – Number of external threads that will use the PushRows APIs
- **omp_max_threads** – Maximum number of OpenMP threads (-1 for default)

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetMarkFinished**(*DatasetHandle* dataset)

Mark the Dataset as complete by calling `dataset->FinishLoad`.

Parameters

- **dataset** – Handle of dataset

Returns

0 when succeed, -1 when failure happens

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

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

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

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

Parameters

- **dataset** – Handle of dataset
- **indptr** – Pointer to row headers
- **indptr_type** – Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- **indices** – Pointer to column indices
- **data** – Pointer to the data space
- **data_type** – Type of data pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`
- **nindptr** – Number of rows in the matrix + 1
- **nelem** – Number of nonzero elements in the matrix

- **num_col** – Number of columns
- **start_row** – Row start index

Returns

0 when succeed, -1 when failure happens

```
LIGHTGBM_C_EXPORT int LGBM_DatasetPushRowsByCSRWithMetadata(DatasetHandle dataset, const void
                                                               *indptr, int indptr_type, const
                                                               int32_t *indices, const void *data,
                                                               int data_type, int64_t nindptr,
                                                               int64_t nelem, int64_t start_row,
                                                               const float *label, const float
                                                               *weight, const double *init_score,
                                                               const int32_t *query, int32_t tid)
```

Push CSR data to existing dataset. (See `LGBM_DatasetPushRowsWithMetadata` for more details.)

Parameters

- **dataset** – Handle of dataset
- **indptr** – Pointer to row headers
- **indptr_type** – Type of `indptr`, can be `C_API_DTYPE_INT32` or `C_API_DTYPE_INT64`
- **indices** – Pointer to column indices
- **data** – Pointer to the data space
- **data_type** – Type of `data` pointer, can be `C_API_DTYPE_FLOAT32` or `C_API_DTYPE_FLOAT64`
- **nindptr** – Number of rows in the matrix + 1
- **nelem** – Number of nonzero elements in the matrix
- **start_row** – Row start index
- **label** – Pointer to array with `nindptr-1` labels
- **weight** – Optional pointer to array with `nindptr-1` weights
- **init_score** – Optional pointer to array with `(nindptr-1)*nclasses` initial scores, in column format
- **query** – Optional pointer to array with `nindptr-1` query values
- **tid** – The id of the calling thread, from 0...N-1 threads

Returns

0 when succeed, -1 when failure happens

```
LIGHTGBM_C_EXPORT int LGBM_DatasetPushRowsWithMetadata(DatasetHandle dataset, const void *data,
                                                               int data_type, int32_t nrow, int32_t ncol,
                                                               int32_t start_row, const float *label, const
                                                               float *weight, const double *init_score,
                                                               const int32_t *query, int32_t tid)
```

Push data to existing dataset. The general flow for a streaming scenario is:

- a. create Dataset “schema” (e.g. `LGBM_DatasetCreateFromSampledColumn`)
- b. init them for thread-safe streaming (`LGBM_DatasetInitStreaming`)

- c. push data (LGBM_DatasetPushRowsWithMetadata or LGBM_DatasetPushRowsByCSRWithMetadata)
- d. call LGBM_DatasetMarkFinished

Parameters

- **dataset** – Handle of dataset
- **data** – Pointer to the data space
- **data_type** – Type of data pointer, can be C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64
- **nrow** – Number of rows
- **ncol** – Number of feature columns
- **start_row** – Row start index, i.e., the index at which to start inserting data
- **label** – Pointer to array with nrow labels
- **weight** – Optional pointer to array with nrow weights
- **init_score** – Optional pointer to array with nrow*nclasses initial scores, in column format
- **query** – Optional pointer to array with nrow query values
- **tid** – The id of the calling thread, from 0...N-1 threads

Returns

0 when succeed, -1 when failure happens

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

Save dataset to binary file.

Parameters

- **handle** – Handle of dataset
- **filename** – The name of the file

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetSerializeReferenceToBinary**(*DatasetHandle* handle,
ByteBufferHandle *out, int32_t
*out_len)

Create a dataset schema representation as a binary byte array (excluding data).

Parameters

- **handle** – Handle of dataset
- **out** – [out] The output byte array
- **out_len** – [out] The length of the output byte array (returned for convenience)

Returns

0 when succeed, -1 when failure happens

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

Save feature names to dataset.

Parameters

- **handle** – Handle of dataset
- **feature_names** – Feature names
- **num_feature_names** – Number of feature names

Returns

0 when succeed, -1 when failure happens

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

Set vector to a content in info.

Note

- *group* only works for C_API_DTYPE_INT32;
- *label* and *weight* only work for C_API_DTYPE_FLOAT32;
- *init_score* only works for C_API_DTYPE_FLOAT64.

Parameters

- **handle** – Handle of dataset
- **field_name** – Field name, can be *label*, *weight*, *init_score*, *group*
- **field_data** – Pointer to data vector
- **num_element** – Number of elements in **field_data**
- **type** – Type of **field_data** pointer, can be C_API_DTYPE_INT32, C_API_DTYPE_FLOAT32 or C_API_DTYPE_FLOAT64

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetSetFieldFromArrow**(*DatasetHandle* handle, const char *field_name, int64_t n_chunks, const struct ArrowArray *chunks, const struct ArrowSchema *schema)

Set vector to a content in info.

Note

- *group* converts input datatype into int32;
- *label* and *weight* convert input datatype into float32;
- *init_score* converts input datatype into float64.

Parameters

- **handle** – Handle of dataset
- **field_name** – Field name, can be *label*, *weight*, *init_score*, *group*
- **n_chunks** – The number of Arrow arrays passed to this function

- **chunks** – Pointer to the list of Arrow arrays
- **schema** – Pointer to the schema of all Arrow arrays

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DatasetSetWaitForManualFinish**(*DatasetHandle* dataset, int wait)

Set whether or not the Dataset waits for a manual MarkFinished call or calls FinishLoad on itself automatically.

Set to 1 for streaming scenario, and use **LGBM_DatasetMarkFinished** to manually finish the Dataset.

Parameters

- **dataset** – Handle of dataset
- **wait** – Whether to wait or not (1 or 0)

Returns

0 when succeed, -1 when failure happens

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

Raise errors for attempts to update dataset parameters.

Parameters

- **old_parameters** – Current dataset parameters
- **new_parameters** – New dataset parameters

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_DumpParamAliases**(int64_t buffer_len, int64_t *out_len, char *out_str)

Dump all parameter names with their aliases to JSON.

Parameters

- **buffer_len** – String buffer length, if **buffer_len** < **out_len**, you should re-allocate buffer
- **out_len** – [out] Actual output length
- **out_str** – [out] JSON format string of parameters, should pre-allocate memory

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_FastConfigFree**(*FastConfigHandle* fastConfig)

Release FastConfig object.

Parameters

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

Returns

0 when it succeeds, -1 when failure happens

LIGHTGBM_C_EXPORT const char ***LGBM_GetLastError**()

Get string message of the last error.

Returns

Error information

LIGHTGBM_C_EXPORT int **LGBM_GetMaxThreads**(int *out)

Get current maximum number of threads used by LightGBM routines in this process.

Parameters

- **out** – [out] current maximum number of threads used by LightGBM. -1 means defaulting to `omp_get_num_threads()`.

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_GetSampleCount**(int32_t num_total_row, const char *parameters, int *out)

Get number of samples based on parameters and total number of rows of data.

Parameters

- **num_total_row** – Number of total rows
- **parameters** – Additional parameters, namely, `bin_construct_sample_cnt` is used to calculate returned value
- **out** – [out] Number of samples. This value is used to pre-allocate memory to hold sample indices when calling `LGBM_SampleIndices`

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_NetworkFree**()

Finalize the network.

Returns

0 when succeed, -1 when failure happens

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

Initialize the network.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Initialize the network with external collective functions.

Parameters

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

Returns

0 when succeed, -1 when failure happens

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

Register a callback function for log redirecting.

Parameters

- **callback** – The callback function to register

Returns

0 when succeed, -1 when failure happens

LIGHTGBM_C_EXPORT int **LGBM_SampleIndices**(int32_t num_total_row, const char *parameters, void *out, int32_t *out_len)

Create sample indices for total number of rows.

Note

You should pre-allocate memory for **out**, you can get its length by **LGBM_GetSampleCount**.

Parameters

- **num_total_row** – Number of total rows
- **parameters** – Additional parameters, namely, `bin_construct_sample_cnt` and `data_random_seed` are used to produce the output
- **out** – **[out]** Created indices, type is `int32_t`
- **out_len** – **[out]** Number of indices

Returns

0 when succeed, -1 when failure happens

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

Set string message of the last error.

Note

This will call unsafe `sprintf` when compiled using C standards before C99.

Parameters

- **msg** – Error message

LIGHTGBM_C_EXPORT int **LGBM_SetMaxThreads**(int num_threads)

Set maximum number of threads used by LightGBM routines in this process.

Parameters

- **num_threads** – maximum number of threads used by LightGBM. -1 means defaulting to `omp_get_num_threads()`.

Returns

0 when succeed, -1 when failure happens

PYTHON API

9.1 Data Structure API

<code>Dataset</code> (data[, label, reference, weight, ...])	Dataset in LightGBM.
<code>Booster</code> ([params, train_set, model_file, ...])	Booster in LightGBM.
<code>CVBooster</code> ([model_file])	CVBooster in LightGBM.
<code>Sequence</code> ()	Generic data access interface.

9.1.1 `lightgbm.Dataset`

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

Bases: `object`

Dataset in LightGBM.

LightGBM does not train on raw data. It discretizes continuous features into histogram bins, tries to combine categorical features, and automatically handles missing and infinite values.

This class handles that preprocessing, and holds that alternative representation of the input data.

```
__init__(data, label=None, reference=None, weight=None, group=None, init_score=None,  
        feature_name='auto', categorical_feature='auto', params=None, free_raw_data=True,  
        position=None)
```

Initialize Dataset.

Parameters

- **data** (`str, pathlib.Path, numpy array, pandas DataFrame, scipy.sparse, Sequence, list of Sequence, list of numpy array or pyarrow Table`) – Data source of Dataset. If str or `pathlib.Path`, it represents the path to a text file (CSV, TSV, or LibSVM) or a LightGBM Dataset binary file.
- **label** (`list, numpy 1-D array, pandas Series / one-column DataFrame, pyarrow Array, pyarrow ChunkedArray 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, pyarrow Array, pyarrow ChunkedArray or None, optional (default=None)`) – Weight for each instance. Weights should be non-negative.

- **group** (*list, numpy 1-D array, pandas Series, pyarrow Array, pyarrow ChunkedArray or None, optional (default=None)*) – Group/query data. Only used in the learning-to-rank task. $\text{sum}(\text{group}) = \text{n_samples}$. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **init_score** (*list, list of lists (for multi-class task), numpy array, pandas Series, pandas DataFrame (for multi-class task), pyarrow Array, pyarrow ChunkedArray, pyarrow Table (for multi-class task) or None, optional (default=None)*) – Init score for Dataset.
- **feature_name** (*list of str, or 'auto', optional (default="auto")*) – Feature names. If 'auto' and data is pandas DataFrame or pyarrow Table, data columns names are used.
- **categorical_feature** (*list of str or int, or 'auto', optional (default="auto")*) – Categorical features. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify `feature_name` as well). If 'auto' and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.
- **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.
- **position** (*numpy 1-D array, pandas Series or None, optional (default=None)*) – Position of items used in unbiased learning-to-rank task.

Methods

<code>__init__(data[, label, reference, weight, ...])</code>	Initialize Dataset.
<code>add_features_from(other)</code>	Add features from other Dataset to the current Dataset.
<code>construct()</code>	Lazy init.
<code>create_valid(data[, label, weight, group, ...])</code>	Create validation data align with current Dataset.
<code>feature_num_bin(feature)</code>	Get the number of bins for a feature.
<code>get_data()</code>	Get the raw data of the Dataset.
<code>get_feature_name()</code>	Get the names of columns (features) in the Dataset.
<code>get_field(field_name)</code>	Get property from the Dataset.
<code>get_group()</code>	Get the group of the Dataset.
<code>get_init_score()</code>	Get the initial score of the Dataset.
<code>get_label()</code>	Get the label of the Dataset.
<code>get_params()</code>	Get the used parameters in the Dataset.
<code>get_position()</code>	Get the position of the Dataset.
<code>get_ref_chain([ref_limit])</code>	Get a chain of Dataset objects.
<code>get_weight()</code>	Get the weight of the Dataset.
<code>num_data()</code>	Get the number of rows in the Dataset.

continues on next page

Table 2 – continued from previous page

<code>num_feature()</code>	Get the number of columns (features) in the Dataset.
<code>save_binary(filename)</code>	Save Dataset to a binary file.
<code>set_categorical_feature(categorical_feature)</code>	Set categorical features.
<code>set_feature_name(feature_name)</code>	Set feature name.
<code>set_field(field_name, data)</code>	Set property into the Dataset.
<code>set_group(group)</code>	Set group size of Dataset (used for ranking).
<code>set_init_score(init_score)</code>	Set init score of Booster to start from.
<code>set_label(label)</code>	Set label of Dataset.
<code>set_position(position)</code>	Set position of Dataset (used for ranking).
<code>set_reference(reference)</code>	Set reference Dataset.
<code>set_weight(weight)</code>	Set weight of each instance.
<code>subset(used_indices[, params])</code>	Get subset of current Dataset.

`add_features_from(other)`

Add features from other Dataset to the current Dataset.

Both Datasets must be constructed before calling this method.

Parameters

`other` ([Dataset](#)) – The Dataset to take features from.

Returns

`self` – Dataset with the new features added.

Return type

[Dataset](#)

`construct()`

Lazy init.

Returns

`self` – Constructed Dataset object.

Return type

[Dataset](#)

`create_valid(data, label=None, weight=None, group=None, init_score=None, params=None, position=None)`

Create validation data align with current Dataset.

Parameters

- **data** (`str, pathlib.Path, numpy array, pandas DataFrame, scipy.sparse, Sequence, list of Sequence, list of numpy array or pyarrow Table`) – Data source of Dataset. If str or pathlib.Path, it represents the path to a text file (CSV, TSV, or LibSVM) or a LightGBM Dataset binary file.
- **label** (`list, numpy 1-D array, pandas Series / one-column DataFrame, pyarrow Array, pyarrow ChunkedArray or None, optional (default=None)`) – Label of the data.
- **weight** (`list, numpy 1-D array, pandas Series, pyarrow Array, pyarrow ChunkedArray or None, optional (default=None)`) – Weight for each instance. Weights should be non-negative.
- **group** (`list, numpy 1-D array, pandas Series, pyarrow Array, pyarrow ChunkedArray or None, optional (default=None)`) – Group/query data. Only used in the learning-to-rank task. $\text{sum}(\text{group}) = \text{n_samples}$. For example, if you have a

100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

- `init_score` (*list, list of lists (for multi-class task), numpy array, pandas Series, pandas DataFrame (for multi-class task), pyarrow Array, pyarrow ChunkedArray, pyarrow Table (for multi-class task) or None, optional (default=None)*) – Init score for Dataset.
- `params` (*dict or None, optional (default=None)*) – Other parameters for validation Dataset.
- `position` (*numpy 1-D array, pandas Series or None, optional (default=None)*) – Position of items used in unbiased learning-to-rank task.

Returns

`valid` – Validation Dataset with reference to self.

Return type

Dataset

feature_num_bin(feature)

Get the number of bins for a feature.

Added in version 4.0.0.

Parameters

`feature` (*int or str*) – Index or name of the feature.

Returns

`number_of_bins` – The number of constructed bins for the feature in the Dataset.

Return type

int

get_data()

Get the raw data of the Dataset.

Returns

`data` – Raw data used in the Dataset construction.

Return type

str, pathlib.Path, numpy array, pandas DataFrame, scipy.sparse, Sequence, list of Sequence, list of numpy array, pyarrow Table or None

get_feature_name()

Get the names of columns (features) in the Dataset.

Returns

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

Return type

list of str

get_field(field_name)

Get property from the Dataset.

Can only be run on a constructed Dataset.

Unlike `get_group()`, `get_init_score()`, `get_label()`, `get_position()`, and `get_weight()`, this method ignores any raw data passed into `lgb.Dataset()` on the Python side, and will only read data from the constructed C++ Dataset object.

Parameters

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

Returns

info – A numpy array with information from the Dataset.

Return type

numpy array or None

get_group()

Get the group of the Dataset.

Returns

group – Group/query data. Only used in the learning-to-rank task. $\text{sum}(\text{group}) = \text{n_samples}$.

For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc. For a constructed `Dataset`, this will only return `None` or a numpy array.

Return type

list, numpy 1-D array, pandas Series, pyarrow Array, pyarrow ChunkedArray or None

get_init_score()

Get the initial score of the Dataset.

Returns

init_score – Init score of Booster. For a constructed `Dataset`, this will only return `None` or a numpy array.

Return type

list, list of lists (for multi-class task), numpy array, pandas Series, pandas DataFrame (for multi-class task), pyarrow Array, pyarrow ChunkedArray, pyarrow Table (for multi-class task) or `None`

get_label()

Get the label of the Dataset.

Returns

label – The label information from the Dataset. For a constructed `Dataset`, this will only return a numpy array.

Return type

list, numpy 1-D array, pandas Series / one-column DataFrame, pyarrow Array, pyarrow ChunkedArray or `None`

get_params()

Get the used parameters in the Dataset.

Returns

params – The used parameters in this `Dataset` object.

Return type

dict

get_position()

Get the position of the Dataset.

Returns

position – Position of items used in unbiased learning-to-rank task. For a constructed `Dataset`, this will only return `None` or a numpy array.

Return type

numpy 1-D array, pandas Series 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. Weights should be non-negative. For a constructed Dataset, this will only return None or a numpy array.

Return type

list, numpy 1-D array, pandas Series, pyarrow Array, pyarrow ChunkedArray or None

num_data()

Get the number of rows in the Dataset.

Returns

number_of_rows – The number of rows in the Dataset.

Return type

int

num_feature()

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

Returns

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

Return type

int

save_binary(*filename*)

Save Dataset to a binary file.

Note

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

Parameters

filename (str or *pathlib.Path*) – 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 str or int, or 'auto'*) – 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 str*) – 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**(*str*) – The field name of the information.
- **data** (*list, list of lists (for multi-class task), numpy array, pandas Series, pandas DataFrame (for multi-class task), pyarrow Array, pyarrow ChunkedArray, pyarrow Table (for multi-class task) or None*) – The 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, pyarrow Array, pyarrow ChunkedArray or None*) – Group/query data. Only used in the learning-to-rank task. $\text{sum}(\text{group}) = \text{n_samples}$. For example, if you have a 100-document dataset with $\text{group} = [10, 20, 40, 10, 10, 10]$, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

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*, *list of lists* (for multi-class task), *numpy array*, *pandas Series*, *pandas DataFrame* (for multi-class task), *pyarrow Array*, *pyarrow ChunkedArray*, *pyarrow Table* (for multi-class task) 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*, *pyarrow Array*, *pyarrow ChunkedArray* or *None*) – The label information to be set into Dataset.

Returns

self – Dataset with set label.

Return type

Dataset

set_position(*position*)

Set position of Dataset (used for ranking).

Parameters

position (*numpy 1-D array*, *pandas Series* or *None*, optional (*default=None*)) – Position of items used in unbiased learning-to-rank task.

Returns

self – Dataset with set position.

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*, *pyarrow Array*, *pyarrow ChunkedArray* or *None*) – Weight to be set for each data point. Weights should be non-negative.

Returns`self` – Dataset with set weight.**Return type**`Dataset`**subset**(*used_indices*, *params*=*None*)

Get subset of current Dataset.

Parameters

- **used_indices** (*list of int*) – Indices used to create the subset.
- **params** (*dict or None, optional (default=None)*) – These parameters will be passed to Dataset constructor.

Returns`subset` – Subset of the current Dataset.**Return type**`Dataset`

9.1.2 `lightgbm.Booster`

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

Booster in LightGBM.

__init__(*params*=*None*, *train_set*=*None*, *model_file*=*None*, *model_str*=*None*)

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** (*str, pathlib.Path or None, optional (default=None)*) – Path to the model file.
- **model_str** (*str or None, optional (default=None)*) – Model will be loaded from this string.

Methods

<code>__init__</code> ([<i>params</i> , <i>train_set</i> , <i>model_file</i> , ...])	Initialize the Booster.
<code>add_valid</code> (<i>data</i> , <i>name</i>)	Add validation data.
<code>current_iteration</code> ()	Get the index of the current iteration.
<code>dump_model</code> ([<i>num_iteration</i> , <i>start_iteration</i> , ...])	Dump Booster to JSON format.
<code>eval</code> (<i>data</i> , <i>name</i> [, <i>feval</i>])	Evaluate for data.
<code>eval_train</code> ([<i>feval</i>])	Evaluate for training data.
<code>eval_valid</code> ([<i>feval</i>])	Evaluate for validation data.
<code>feature_importance</code> ([<i>importance_type</i> , <i>iteration</i>])	Get feature importances.
<code>feature_name</code> ()	Get names of features.
<code>free_dataset</code> ()	Free Booster's Datasets.
<code>free_network</code> ()	Free Booster's network.
<code>get_leaf_output</code> (<i>tree_id</i> , <i>leaf_id</i>)	Get the output of a leaf.

continues on next page

Table 3 – continued from previous page

<code>get_split_value_histogram(feature[, bins, ...])</code>	Get split value histogram for the specified feature.
<code>lower_bound()</code>	Get lower bound value of a model.
<code>model_from_string(model_str)</code>	Load Booster from a string.
<code>model_to_string([num_iteration, ...])</code>	Save Booster to string.
<code>num_feature()</code>	Get number of features.
<code>num_model_per_iteration()</code>	Get number of models per iteration.
<code>num_trees()</code>	Get number of weak sub-models.
<code>predict(data[, start_iteration, ...])</code>	Make a prediction.
<code>refit(data, label[, decay_rate, reference, ...])</code>	Refit the existing Booster by new data.
<code>reset_parameter(params)</code>	Reset parameters of Booster.
<code>rollback_one_iter()</code>	Rollback one iteration.
<code>save_model(filename[, num_iteration, ...])</code>	Save Booster to file.
<code>set_leaf_output(tree_id, leaf_id, value)</code>	Set the output of a leaf.
<code>set_network(machines[, local_listen_port, ...])</code>	Set the network configuration.
<code>set_train_data_name(name)</code>	Set the name to the training Dataset.
<code>shuffle_models([start_iteration, end_iteration])</code>	Shuffle models.
<code>trees_to_dataframe()</code>	Parse the fitted model and return in an easy-to-read pandas DataFrame.
<code>update([train_set, fobj])</code>	Update Booster for one iteration.
<code>upper_bound()</code>	Get upper bound value of a model.

add_valid(data, name)

Add validation data.

Parameters

- **data** ([Dataset](#)) – Validation data.
- **name** (*str*) – Name of validation data.

Returns

`self` – Booster with set validation data.

Return type

[Booster](#)

current_iteration()

Get the index of the current iteration.

Returns

`cur_iter` – The index of the current iteration.

Return type

`int`

dump_model(num_iteration=None, start_iteration=0, importance_type='split', object_hook=None)

Dump Booster to JSON format.

Parameters

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

- **importance_type** (*str, optional (default="split")*) – What type of feature importance should be dumped. If “split”, result contains numbers of times the feature is used in a model. If “gain”, result contains total gains of splits which use the feature.
- **object_hook** (*callable or None, optional (default=None)*) – If not None, `object_hook` is a function called while parsing the json string returned by the C API. It may be used to alter the json, to store specific values while building the json structure. It avoids walking through the structure again. It saves a significant amount of time if the number of trees is huge. Signature is `def object_hook(node: dict) -> dict`. `None` is equivalent to `lambda node: node`. See documentation of `json.loads()` for further details.

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** (*str*) – Name of the data.
- **feval** (*callable, list of callable, or None, optional (default=None)*) – Customized evaluation function. Each evaluation function should accept two parameters: `preds`, `eval_data`, and return (`eval_name`, `eval_result`, `is_higher_better`) or list of such tuples.

preds

[numpy 1-D array or numpy 2-D array (for multi-class task)] The predicted values. For multi-class task, `preds` are numpy 2-D array of shape = `[n_samples, n_classes]`. If custom objective function is used, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

eval_data

[`Dataset`] A `Dataset` to evaluate.

eval_name

[*str*] The name of evaluation function (without whitespace).

eval_result

[*float*] The eval result.

is_higher_better

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

Returns

`result` – List with (`dataset_name`, `eval_name`, `eval_result`, `is_higher_better`) tuples.

Return type

list

eval_train (*feval=None*)

Evaluate for training data.

Parameters

- **feval** (*callable, list of callable, or None, optional (default=None)*)

- Customized evaluation function. Each evaluation function should accept two parameters: preds, eval_data, and return (eval_name, eval_result, is_higher_better) or list of such tuples.

preds

[numpy 1-D array or numpy 2-D array (for multi-class task)] The predicted values. For multi-class task, preds are numpy 2-D array of shape = [n_samples, n_classes]. If custom objective function is used, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

eval_data

[Dataset] The training dataset.

eval_name

[str] The name of evaluation function (without whitespace).

eval_result

[float] The eval result.

is_higher_better

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

Returns

result – List with (train_dataset_name, eval_name, eval_result, is_higher_better) tuples.

Return type

list

eval_valid(feval=None)

Evaluate for validation data.

Parameters

feval (callable, list of callable, or None, optional (default=None))

- Customized evaluation function. Each evaluation function should accept two parameters: preds, eval_data, and return (eval_name, eval_result, is_higher_better) or list of such tuples.

preds

[numpy 1-D array or numpy 2-D array (for multi-class task)] The predicted values. For multi-class task, preds are numpy 2-D array of shape = [n_samples, n_classes]. If custom objective function is used, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

eval_data

[Dataset] The validation dataset.

eval_name

[str] The name of evaluation function (without whitespace).

eval_result

[float] The eval result.

is_higher_better

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

Returns

result – List with (validation_dataset_name, eval_name, eval_result, is_higher_better) tuples.

Return type

list

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

Get feature importances.

Parameters

- **importance_type**(*str, 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 of str

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

⚠ Warning

Categorical features are not supported.

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

Returns

- **result_tuple (tuple of 2 numpy arrays)** – If xgboost_style=False, the values of the histogram of used splitting values for the specified feature and the bin edges.
- **result_array_like (numpy array or pandas DataFrame (if pandas is installed))** – If xgboost_style=True, the histogram of used splitting values for the specified feature.

lower_bound()

Get lower bound value of a model.

Returns

`lower_bound` – Lower bound value of the model.

Return type

float

model_from_string(model_str)

Load Booster from a string.

Parameters

`model_str (str)` – Model will be loaded from this string.

Returns

`self` – Loaded Booster object.

Return type

Booster

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

Save Booster to string.

Parameters

- **num_iteration** (*int or None, optional (default=None)*) – Index of the iteration that should be saved. If None, if the best iteration exists, it is saved; otherwise, all iterations are saved. If $<= 0$, all iterations are saved.
- **start_iteration** (*int, optional (default=0)*) – Start index of the iteration that should be saved.
- **importance_type** (*str, optional (default="split")*) – What type of feature importance should be saved. If “split”, result contains numbers of times the feature is used in a model. If “gain”, result contains total gains of splits which use the feature.

Returns

str_repr – String representation of Booster.

Return type

str

num_feature()

Get number of features.

Returns

num_feature – The number of features.

Return type

int

num_model_per_iteration()

Get number of models per iteration.

Returns

model_per_iter – The number of models per iteration.

Return type

int

num_trees()

Get number of weak sub-models.

Returns

num_trees – The number of weak sub-models.

Return type

int

predict(*data, start_iteration=0, num_iteration=None, raw_score=False, pred_leaf=False, pred_contrib=False, data_has_header=False, validate_features=False, **kwargs*)

Make a prediction.

Parameters

- **data** (*str, pathlib.Path, numpy array, pandas DataFrame, scipy.sparse or pyarrow Table*) – Data source for prediction. If str or pathlib.Path, it represents the path to a text file (CSV, TSV, or LibSVM).
- **start_iteration** (*int, optional (default=0)*) – Start index of the iteration to predict. If $<= 0$, starts from the first iteration.
- **num_iteration** (*int or None, optional (default=None)*) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration $<= 0$, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If $<= 0$, all iterations from start_iteration are used (no limits).

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

Note

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

- **data_has_header** (bool, optional (default=False)) – Whether the data has header. Used only if data is str.
- **validate_features** (bool, optional (default=False)) – If True, ensure that the features used to predict match the ones used to train. Used only if data is pandas DataFrame.
- ****kwargs** – Other parameters for the prediction.

Returns

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

Return type

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

refit(*data*, *label*, *decay_rate*=0.9, *reference*=None, *weight*=None, *group*=None, *init_score*=None, *feature_name*='auto', *categorical_feature*='auto', *dataset_params*=None, *free_raw_data*=True, *validate_features*=False, ****kwargs**)

Refit the existing Booster by new data.

Parameters

- **data** (str, `pathlib.Path`, numpy array, pandas DataFrame, `scipy.sparse`, `Sequence`, list of `Sequence`, list of numpy array or `pyarrow Table`) – Data source for refit. If str or `pathlib.Path`, it represents the path to a text file (CSV, TSV, or LibSVM).
- **label** (list, numpy 1-D array, pandas Series / one-column DataFrame, `pyarrow Array` or `pyarrow ChunkedArray`) – 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.
- **reference** (`Dataset` or None, optional (default=None)) – Reference for data.

Added in version 4.0.0.

- **weight** (list, numpy 1-D array, pandas Series, `pyarrow Array`, `pyarrow ChunkedArray` or None, optional (default=None)) – Weight for each data instance. Weights should be non-negative.

Added in version 4.0.0.

- **group** (*list, numpy 1-D array, pandas Series, pyarrow Array, pyarrow ChunkedArray or None, optional (default=None)*) – Group/query size for data. Only used in the learning-to-rank task. $\text{sum}(\text{group}) = \text{n_samples}$. For example, if you have a 100-document dataset with $\text{group} = [10, 20, 40, 10, 10, 10]$, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

Added in version 4.0.0.

- **init_score** (*list, list of lists (for multi-class task), numpy array, pandas Series, pandas DataFrame (for multi-class task), pyarrow Array, pyarrow ChunkedArray, pyarrow Table (for multi-class task) or None, optional (default=None)*) – Init score for data.

Added in version 4.0.0.

- **feature_name** (*list of str, or 'auto', optional (default="auto")*) – Feature names for data. If ‘auto’ and data is pandas DataFrame, data columns names are used.

Added in version 4.0.0.

- **categorical_feature** (*list of str or int, or 'auto', optional (default="auto")*) – Categorical features for data. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify **feature_name** as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.

Added in version 4.0.0.

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

Added in version 4.0.0.

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

Added in version 4.0.0.

- **validate_features** (*bool, optional (default=False)*) – If True, ensure that the features used to refit the model match the original ones. Used only if data is pandas DataFrame.

Added in version 4.0.0.

- ****kwargs** – Other parameters for refit. These parameters will be passed to **predict** method.

Returns

result – Refitted Booster.

Return type*Booster***reset_parameter**(*params*)

Reset parameters of Booster.

Parameters

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

Returns

self – Booster with new parameters.

Return type*Booster***rollback_one_iter**()

Rollback one iteration.

Returns

self – Booster with rolled back one iteration.

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

Save Booster to file.

Parameters

- **filename** (*str* or *pathlib.Path*) – Filename to save Booster.
- **num_iteration** (*int* or *None*, *optional* (*default=None*)) – Index of the iteration that should be saved. If *None*, if the best iteration exists, it is saved; otherwise, all iterations are saved. If $<= 0$, all iterations are saved.
- **start_iteration** (*int*, *optional* (*default=0*)) – Start index of the iteration that should be saved.
- **importance_type** (*str*, *optional* (*default="split"*)) – What type of feature importance should be saved. If “split”, result contains numbers of times the feature is used in a model. If “gain”, result contains total gains of splits which use the feature.

Returns

self – Returns self.

Return type*Booster***set_leaf_output**(*tree_id*, *leaf_id*, *value*)

Set the output of a leaf.

Added in version 4.0.0.

Parameters

- **tree_id** (*int*) – The index of the tree.
- **leaf_id** (*int*) – The index of the leaf in the tree.
- **value** (*float*) – Value to set as the output of the leaf.

Returns

self – Booster with the leaf output set.

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 str*) – Names of machines.
- **local_listen_port** (*int, optional (default=12400)*) – TCP listen port for local machines.
- **listen_time_out** (*int, optional (default=120)*) – Socket time-out in minutes.
- **num_machines** (*int, optional (default=1)*) – The number of machines for distributed learning application.

Returns**self** – Booster with set network.**Return type***Booster***set_train_data_name**(*name*)

Set the name to the training Dataset.

Parameters

- **name** (*str*) – Name for the training Dataset.

Returns**self** – Booster with set training Dataset name.**Return type***Booster***shuffle_models**(*start_iteration*=0, *end_iteration*=-1)

Shuffle models.

Parameters

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

Returns**self** – Booster with shuffled models.**Return type***Booster***trees_to_dataframe**()

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

The returned DataFrame has the following columns.

- **tree_index** : int64, which tree a node belongs to. 0-based, so a value of 6, for example, means “this node is in the 7th tree”.
- **node_depth** : int64, how far a node is from the root of the tree. The root node has a value of 1, its direct children are 2, etc.

- `node_index` : str, unique identifier for a node.
- `left_child` : str, `node_index` of the child node to the left of a split. `None` for leaf nodes.
- `right_child` : str, `node_index` of the child node to the right of a split. `None` for leaf nodes.
- `parent_index` : str, `node_index` of this node's parent. `None` for the root node.
- `split_feature` : str, name of the feature used for splitting. `None` for leaf nodes.
- `split_gain` : float64, gain from adding this split to the tree. `NaN` for leaf nodes.
- `threshold` : float64, value of the feature used to decide which side of the split a record will go down. `NaN` for leaf nodes.
- `decision_type` : str, logical operator describing how to compare a value to `threshold`. For example, `split_feature = "Column_10"`, `threshold = 15`, `decision_type = "<="` means that records where `Column_10 <= 15` follow the left side of the split, otherwise follows the right side of the split. `None` for leaf nodes.
- `missing_direction` : str, split direction that missing values should go to. `None` for leaf nodes.
- `missing_type` : str, describes what types of values are treated as missing.
- `value` : float64, predicted value for this leaf node, multiplied by the learning rate.
- `weight` : float64 or int64, sum of Hessian (second-order derivative of objective), summed over observations that fall in this node.
- `count` : int64, number of records in the training data that fall into this node.

Returns

`result` – Returns a pandas DataFrame of the parsed model.

Return type

pandas DataFrame

update(`train_set=None`, `fobj=None`)

Update Booster for one iteration.

Parameters

- `train_set` (`Dataset` or `None`, optional (`default=None`)) – Training data. If `None`, last training data is used.
- `fobj` (`callable` or `None`, optional (`default=None`)) – Customized objective function. Should accept two parameters: `preds`, `train_data`, and return (`grad`, `hess`).

`preds`

[numpy 1-D array or numpy 2-D array (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

`train_data`

[`Dataset`] The training dataset.

`grad`

[numpy 1-D array or numpy 2-D array (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `preds` for each sample point.

`hess`

[numpy 1-D array or numpy 2-D array (for multi-class task)] The value of the

second order derivative (Hessian) of the loss with respect to the elements of preds for each sample point.

For multi-class task, preds are numpy 2-D array of shape = [n_samples, n_classes], and grad and hess should be returned in the same format.

Returns

produced_empty_tree – True if the tree(s) produced by this iteration did not have any splits. This usually means that training is “finished” (calling `update()` again will not change the model’s predictions). However, that is not always the case. For example, if you have added any randomness (like column sampling by setting `feature_fraction_bynode < 1.0`), it is possible that another call to `update()` would produce a non-empty tree.

Return type

bool

upper_bound()

Get upper bound value of a model.

Returns

upper_bound – Upper bound value of the model.

Return type

float

9.1.3 `lightgbm.CVBooster`

`class lightgbm.CVBooster(model_file=None)`

Bases: `object`

CVBooster in LightGBM.

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

- `model_from_string()`
- `model_to_string()`
- `save_model()`

boosters

The list of underlying fitted models.

Type

list of `Booster`

best_iteration

The best iteration of fitted model.

Type

int

`__init__(model_file=None)`

Initialize the CVBooster.

Parameters

`model_file (str, pathlib.Path or None, optional (default=None))` – Path to the CVBooster model file.

Methods

<code>__init__([model_file])</code>	Initialize the CVBooster.
<code>model_from_string(model_str)</code>	Load CVBooster from a string.
<code>model_to_string([num_iteration, ...])</code>	Save CVBooster to JSON string.
<code>save_model(filename[, num_iteration, ...])</code>	Save CVBooster to a file as JSON text.

`model_from_string(model_str)`

Load CVBooster from a string.

Parameters

`model_str (str)` – Model will be loaded from this string.

Returns

`self` – Loaded CVBooster object.

Return type

`CVBooster`

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

Save CVBooster to JSON string.

Parameters

- `num_iteration (int or None, optional (default=None))` – Index of the iteration that should be saved. If None, if the best iteration exists, it is saved; otherwise, all iterations are saved. If ≤ 0 , all iterations are saved.
- `start_iteration (int, optional (default=0))` – Start index of the iteration that should be saved.
- `importance_type (str, optional (default="split"))` – What type of feature importance should be saved. If “split”, result contains numbers of times the feature is used in a model. If “gain”, result contains total gains of splits which use the feature.

Returns

`str_repr` – JSON string representation of CVBooster.

Return type

`str`

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

Save CVBooster to a file as JSON text.

Parameters

- `filename (str or pathlib.Path)` – Filename to save CVBooster.
- `num_iteration (int or None, optional (default=None))` – Index of the iteration that should be saved. If None, if the best iteration exists, it is saved; otherwise, all iterations are saved. If ≤ 0 , all iterations are saved.
- `start_iteration (int, optional (default=0))` – Start index of the iteration that should be saved.
- `importance_type (str, optional (default="split"))` – What type of feature importance should be saved. If “split”, result contains numbers of times the feature is used in a model. If “gain”, result contains total gains of splits which use the feature.

Returns

`self` – Returns self.

Return type
CVBooster

9.1.4 `lightgbm.Sequence`

`class lightgbm.Sequence`

Bases: ABC

Generic data access interface.

Object should support the following operations:

```
# Get total row number.
>>> len(seq)
# Random access by row index. Used for data sampling.
>>> seq[10]
# Range data access. Used to read data in batch when constructing Dataset.
>>> seq[0:100]
# Optionally specify batch_size to control range data read size.
>>> seq.batch_size
```

- With random access, **data sampling does not need to go through all data**.
- With range data access, there's **no need to read all data into memory thus reduce memory usage**.

Added in version 3.3.0.

batch_size

Default size of a batch.

Type

int

`__init__()`

Methods

`__init__()`

Attributes

`batch_size`

9.2 Training API

`train`(params, train_set[, num_boost_round, ...])
`cv`(params, train_set[, num_boost_round, ...])

Perform the training with given parameters.
 Perform the cross-validation with given parameters.

9.2.1 lightgbm.train

```
lightgbm.train(params, train_set, num_boost_round=100, valid_sets=None, valid_names=None, feval=None, init_model=None, keep_training_booster=False, callbacks=None)
```

Perform the training with given parameters.

Parameters

- **params** (*dict*) – Parameters for training. Values passed through `params` take precedence over those supplied via arguments.
- **train_set** (`Dataset`) – Data to be trained on.
- **num_boost_round** (*int, optional (default=100)*) – Number of boosting iterations.
- **valid_sets** (*list of Dataset, or None, optional (default=None)*) – List of data to be evaluated on during training.
- **valid_names** (*list of str, or None, optional (default=None)*) – Names of `valid_sets`.
- **feval** (*callable, list of callable, or None, optional (default=None)*) – Customized evaluation function. Each evaluation function should accept two parameters: `preds`, `eval_data`, and return (`eval_name`, `eval_result`, `is_higher_better`) or list of such tuples.

`preds`

[numpy 1-D array or numpy 2-D array (for multi-class task)] The predicted values. For multi-class task, `preds` are numpy 2-D array of shape = [n_samples, n_classes]. If custom objective function is used, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

`eval_data`

[`Dataset`] A `Dataset` to evaluate.

`eval_name`

[`str`] The name of evaluation function (without whitespaces).

`eval_result`

[`float`] The eval result.

`is_higher_better`

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

To ignore the default metric corresponding to the used objective, set the `metric` parameter to the string "None" in `params`.

- **init_model** (*str, pathlib.Path, Booster or None, optional (default=None)*) – Filename of LightGBM model or Booster instance used for continue training.
- **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. This means you won't be able to use `eval`, `eval_train` or `eval_valid` methods of the returned Booster. When your model is very large and cause the memory error, you can try to set this param to `True` to avoid the model conversion performed during the internal call of `model_to_string`. You can still use `_InnerPredictor` as `init_model` for future continue training.

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

 **Note**

A custom objective function can be provided for the `objective` parameter. It should accept two parameters: `preds`, `train_data` and return `(grad, hess)`.

preds

[numpy 1-D array or numpy 2-D array (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

train_data

[Dataset] The training dataset.

grad

[numpy 1-D array or numpy 2-D array (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `preds` for each sample point.

hess

[numpy 1-D array or numpy 2-D array (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of `preds` for each sample point.

For multi-class task, `preds` are numpy 2-D array of shape = `[n_samples, n_classes]`, and `grad` and `hess` should be returned in the same format.

Returns

booster – The trained Booster model.

Return type

Booster

9.2.2 `lightgbm.cv`

```
lightgbm.cv(params, train_set, num_boost_round=100, folds=None, nfold=5, stratified=True, shuffle=True,
            metrics=None, feval=None, init_model=None, fpreproc=None, seed=0, callbacks=None,
            eval_train_metric=False, return_cvbooster=False)
```

Perform the cross-validation with given parameters.

Parameters

- **params** (*dict*) – Parameters for training. Values passed through `params` take precedence over those supplied via arguments.
- **train_set** (*Dataset*) – Data to be trained on.
- **num_boost_round** (*int, optional (default=100)*) – Number of boosting iterations.
- **folds** (*generator or iterator of (train_idx, test_idx) tuples, scikit-learn splitter object or None, optional (default=None)*)
 - If generator or iterator, it should yield the train and test indices for each fold. If object, it should be one of the scikit-learn splitter classes (<https://scikit-learn.org/stable/modules/classes.html#splitter-classes>) and have `split` method. This argument has highest priority over other data split arguments.
- **nfold** (*int, optional (default=5)*) – Number of folds in CV.

- **stratified** (*bool, optional (default=True)*) – Whether to perform stratified sampling.
- **shuffle** (*bool, optional (default=True)*) – Whether to shuffle before splitting data.
- **metrics** (*str, list of str, or None, optional (default=None)*) – Evaluation metrics to be monitored while CV. If not None, the metric in `params` will be overridden.
- **feval** (*callable, list of callable, or None, optional (default=None)*) – Customized evaluation function. Each evaluation function should accept two parameters: `preds`, `eval_data`, and return (`eval_name`, `eval_result`, `is_higher_better`) or list of such tuples.

preds

[numpy 1-D array or numpy 2-D array (for multi-class task)] The predicted values. For multi-class task, `preds` are numpy 2-D array of shape = [n_samples, n_classes]. If custom objective function is used, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

eval_data

[Dataset] A `Dataset` to evaluate.

eval_name

[str] The name of evaluation function (without whitespace).

eval_result

[float] The eval result.

is_higher_better

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

To ignore the default metric corresponding to the used objective, set `metrics` to the string "None".

- **init_model** (*str, pathlib.Path, Booster or None, optional (default=None)*) – Filename of LightGBM model or Booster instance used for continue training.
- **fpreproc** (*callable or None, optional (default=None)*) – Preprocessing function that takes (`dtrain`, `dtest`, `params`) and returns transformed versions of those.
- **seed** (*int, optional (default=0)*) – Seed used to generate the folds (passed to `numpy.random.seed`).
- **callbacks** (*list of callable, or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See `Callbacks` in Python API for more information.
- **eval_train_metric** (*bool, optional (default=False)*) – Whether to display the train metric in progress. The score of the metric is calculated again after each training step, so there is some impact on performance.
- **return_cvbooster** (*bool, optional (default=False)*) – Whether to return Booster models trained on each fold through CVBooster.

Note

A custom objective function can be provided for the `objective` parameter. It should accept two parameters: `preds`, `train_data` and return `(grad, hess)`.

preds

[numpy 1-D array or numpy 2-D array (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

train_data

[Dataset] The training dataset.

grad

[numpy 1-D array or numpy 2-D array (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `preds` for each sample point.

hess

[numpy 1-D array or numpy 2-D array (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of `preds` for each sample point.

For multi-class task, `preds` are numpy 2-D array of shape = `[n_samples, n_classes]`, and `grad` and `hess` should be returned in the same format.

Returns

`eval_results` – History of evaluation results of each metric. The dictionary has the following format: `{'valid metric1-mean': [values], 'valid metric1-stdv': [values], 'valid metric2-mean': [values], 'valid metric2-stdv': [values], ...}`. If `return_cvbooster=True`, also returns trained boosters wrapped in a `CVBooster` object via `cvbooster` key. If `eval_train_metric=True`, also returns the train metric history. In this case, the dictionary has the following format: `{'train metric1-mean': [values], 'valid metric1-mean': [values], 'train metric2-mean': [values], 'valid metric2-mean': [values], ...}`.

Return type

`dict`

9.3 Scikit-learn API

<code>LGBMModel(*[, boosting_type, num_leaves, ...])</code>	Implementation of the scikit-learn API for LightGBM.
<code>LGBMClassifier(*[, boosting_type, ...])</code>	LightGBM classifier.
<code>LGBMRegressor(*[, boosting_type, ...])</code>	LightGBM regressor.
<code>LGBMRanker(*[, boosting_type, num_leaves, ...])</code>	LightGBM ranker.

9.3.1 `lightgbm.LGBMModel`

```
class lightgbm.LGBMModel(*(Keyword-only parameters separator (PEP 3102)), 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=None, importance_type='split',
                         **kwargs)
```

Bases: `BaseEstimator`

Implementation of the scikit-learn API for LightGBM.

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

Construct a gradient boosting model.

Parameters

- **boosting_type** (*str, optional (default='gbdt')*) – ‘gbdt’, traditional Gradient Boosting Decision Tree. ‘dart’, Dropouts meet Multiple Additive Regression Trees. ‘rf’, Random Forest.
- **num_leaves** (*int, optional (default=31)*) – Maximum tree leaves for base learners.
- **max_depth** (*int, optional (default=-1)*) – Maximum tree depth for base learners, $<= 0$ means no limit. If setting this to a positive value, consider also changing `num_leaves` to $<= 2^{\max_depth}$.
- **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** (*str, callable or None, optional (default=None)*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). Default: ‘regression’ for `LGBMRegressor`, ‘binary’ or ‘multiclass’ for `LGBMClassifier`, ‘lambdaRank’ for `LGBMRanker`.
- **class_weight** (*dict, ‘balanced’ or None, optional (default=None)*) – Weights associated with classes in the form `{class_label: weight}`. Use this parameter only for multi-class classification task; for binary classification task you may use `is_unbalance` or `scale_pos_weight` parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (<https://scikit-learn.org/stable/modules/calibration.html>) of your model. The ‘balanced’ mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`. If `None`, all classes are supposed to have weight one. Note, that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.
- **min_split_gain** (*float, optional (default=0.)*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*float, optional (default=1e-3)*) – Minimum sum of instance weight (Hessian) needed in a child (leaf).
- **min_child_samples** (*int, optional (default=20)*) – Minimum number of data needed in a child (leaf).
- **subsample** (*float, optional (default=1.)*) – Subsample ratio of the training instance.

- **subsample_freq** (*int, optional (default=0)*) – Frequency of subsample, $<=0$ means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, RandomState object or None, optional (default=None)*) – Random number seed. If int, this number is used to seed the C++ code. If RandomState or Generator object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.
- **n_jobs** (*int or None, optional (default=None)*) – Number of parallel threads to use for training (can be changed at prediction time by passing it as an extra keyword argument).

For better performance, it is recommended to set this to the number of physical cores in the CPU.

Negative integers are interpreted as following joblib's formula ($n_cpus + 1 + n_jobs$), just like scikit-learn (so e.g. -1 means using all threads). A value of zero corresponds the default number of threads configured for OpenMP in the system. A value of None (the default) corresponds to using the number of physical cores in the system (its correct detection requires either the joblib or the psutil util libraries to be installed).

Changed in version 4.0.0.

- **importance_type** (*str, optional (default='split')*) – The type of feature importance to be filled into `feature_importances_`. If 'split', result contains numbers of times the feature is used in a model. If 'gain', result contains total gains of splits which use the feature.
- ****kwargs** – Other parameters for the model. Check <http://lightgbm.readthedocs.io/en/latest/Parameters.html> for more parameters.

 **Warning**

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

 **Note**

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

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples,

`n_classes]` (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

`weight`

[numpy 1-D array of shape = `[n_samples]`] The weight of samples. Weights should be non-negative.

`group`

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

`grad`

[numpy 1-D array of shape = `[n_samples]` or numpy 2-D array of shape = `[n_samples, n_classes]` (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `y_pred` for each sample point.

`hess`

[numpy 1-D array of shape = `[n_samples]` or numpy 2-D array of shape = `[n_samples, n_classes]` (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of `y_pred` for each sample point.

For multi-class task, `y_pred` is a numpy 2-D array of shape = `[n_samples, n_classes]`, and `grad` and `hess` should be returned in the same format.

Methods

<code>__init__(*, boosting_type, num_leaves, ...)</code>	Construct a gradient boosting model.
<code>fit(X, y[, sample_weight, init_score, ...])</code>	Build a gradient boosting model from the training set <code>(X, y)</code> .
<code>get_metadata_routing()</code>	Get metadata routing of this object.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, raw_score, start_iteration, ...])</code>	Return the predicted value for each sample.
<code>set_fit_request(*[, callbacks, ...])</code>	Configure whether metadata should be requested to be passed to the <code>fit</code> method.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>set_predict_request(*[, num_iteration, ...])</code>	Configure whether metadata should be requested to be passed to the <code>predict</code> method.

Attributes

<code>best_iteration_</code>	The best iteration of fitted model if <code>early_stopping()</code> callback has been specified.
<code>best_score_</code>	The best score of fitted model.
<code>booster_</code>	The underlying Booster of this model.
<code>evals_result_</code>	The evaluation results if validation sets have been specified.
<code>feature_importances_</code>	The feature importances (the higher, the more important).

continues on next page

Table 10 – continued from previous page

<code>feature_name_</code>	The names of features.
<code>feature_names_in_</code>	scikit-learn compatible version of <code>feature_name_</code> .
<code>n_estimators_</code>	True number of boosting iterations performed.
<code>n_features_</code>	The number of features of fitted model.
<code>n_features_in_</code>	The number of features of fitted model.
<code>n_iter_</code>	True number of boosting iterations performed.
<code>objective_</code>	The concrete objective used while fitting this model.

property best_iteration_

The best iteration of fitted model if `early_stopping()` callback has been specified.

Type

`int`

property best_score_

The best score of fitted model.

Type

`dict`

property booster_

The underlying Booster of this model.

Type

`Booster`

property evals_result_

The evaluation results if validation sets have been specified.

Type

`dict`

property feature_importances_

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

Note

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

Type

array of shape = `[n_features]`

property feature_name_

The names of features.

Note

If input does not contain feature names, they will be added during fitting in the format `Column_0`, `Column_1`, ..., `Column_N`.

Type

list of shape = [n_features]

property feature_names_in_

scikit-learn compatible version of .feature_name_.

Added in version 4.5.0.

Type

array of shape = [n_features]

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, feature_name='auto', categorical_feature='auto', callbacks=None, init_model=None)

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

Parameters

- **X** (numpy array, pandas DataFrame, pyarrow Table, scipy.sparse, list of lists of int or float of shape = [n_samples, n_features]) – Input feature matrix.
- **y** (numpy array, pandas DataFrame, pandas Series, list of int or float, pyarrow Array, pyarrow ChunkedArray of shape = [n_samples]) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (numpy array, pandas Series, list of int or float, pyarrow Array, pyarrow ChunkedArray of shape = [n_samples] or None, optional (default=None)) – Weights of training data. Weights should be non-negative.
- **init_score** (numpy array, pandas DataFrame, pandas Series, list of int or float, list of lists, pyarrow Array, pyarrow ChunkedArray, pyarrow Table of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task) or shape = [n_samples, n_classes] (for multi-class task) or None, optional (default=None)) – Init score of training data.
- **group** (numpy array, pandas Series, pyarrow Array, pyarrow ChunkedArray, list of int or float, or None, optional (default=None)) – Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **eval_set** (list or None, optional (default=None)) – A list of (X, y) tuple pairs to use as validation sets.
- **eval_names** (list of str, or None, optional (default=None)) – Names of eval_set.
- **eval_sample_weight** (list of array (same types as sample_weight supports), or None, optional (default=None)) – Weights of eval data. Weights should be non-negative.
- **eval_class_weight** (list or None, optional (default=None)) – Class weights of eval data.

- **eval_init_score** (list of array (same types as `init_score` supports), or None, optional (default=None)) – Init score of eval data.
- **eval_group** (list of array (same types as `group` supports), or None, optional (default=None)) – Group data of eval data.
- **eval_metric** *(str, callable, list or None, optional (default=None))* – If str, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the `metric` from the model parameters will be evaluated and used as well. Default: ‘l2’ for `LGBMRegressor`, ‘logloss’ for `LGBMClassifier`, ‘ndcg’ for `LGBMRanker`.
- **feature_name** *(list of str, or ‘auto’, optional (default='auto'))* – Feature names. If ‘auto’ and data is pandas DataFrame, data columns names are used.
- **categorical_feature** *(list of str or int, or ‘auto’, optional (default='auto'))* – Categorical features. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify `feature_name` as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.
- **callbacks** *(list of callable, or None, optional (default=None))* – List of callback functions that are applied at each iteration. See `Callbacks` in Python API for more information.
- **init_model** *(str, `pathlib.Path`, `Booster`, `LGBMModel` or None, optional (default=None))* – Filename of LightGBM model, `Booster` instance or `LGBMModel` instance used for continue training.

Returns

`self` – Returns self.

Return type

`LGBMModel`

 **Note**

Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns `(eval_name, eval_result, is_higher_better)` or list of `(eval_name, eval_result, is_higher_better)`:

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. In case of `custom objective`, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

weight

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

group

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

eval_name

[str] The name of evaluation function (without whitespace).

eval_result

[float] The eval result.

is_higher_better

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

get_metadata_routing()

Get metadata routing of this object.

Please check [User Guide](#) on how the routing mechanism works.

Returns

`routing` – A `MetadataRequest` encapsulating routing information.

Return type

`MetadataRequest`

get_params(deep=True)

Get parameters for this estimator.

Parameters

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

Returns

`params` – Parameter names mapped to their values.

Return type

`dict`

property n_estimators

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

`int`

property n_features

The number of features of fitted model.

Type

`int`

property n_features_in_

The number of features of fitted model.

Type

int

property n_iter_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property objective_

The concrete objective used while fitting this model.

Type

str or callable

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

Return the predicted value for each sample.

Parameters

- **X** (`numpy array`, `pandas DataFrame`, `scipy.sparse`, `list of lists of int or float` of shape $= [n_samples, n_features]$) – Input features matrix.
- **raw_score** (`bool`, `optional (default=False)`) – Whether to predict raw scores.
- **start_iteration** (`int`, `optional (default=0)`) – Start index of the iteration to predict. If $<= 0$, starts from the first iteration.
- **num_iteration** (`int or None`, `optional (default=None)`) – Total number of iterations used in the prediction. If `None`, if the best iteration exists and `start_iteration` $<= 0$, the best iteration is used; otherwise, all iterations from `start_iteration` are used (no limits). If $<= 0$, all iterations from `start_iteration` are used (no limits).
- **pred_leaf** (`bool`, `optional (default=False)`) – Whether to predict leaf index.
- **pred_contrib** (`bool`, `optional (default=False)`) – Whether to predict feature contributions.

 **Note**

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

- **validate_features** (`bool`, `optional (default=False)`) – If `True`, ensure that the features used to predict match the ones used to train. Used only if data is `pandas DataFrame`.

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

Returns

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

```
set_fit_request(*, callbacks='UNCHANGED$', categorical_feature='UNCHANGED$',  
                eval_class_weight='UNCHANGED$', eval_group='UNCHANGED$',  
                eval_init_score='UNCHANGED$', eval_metric='UNCHANGED$',  
                eval_names='UNCHANGED$', eval_sample_weight='UNCHANGED$',  
                eval_set='UNCHANGED$', feature_name='UNCHANGED$',  
                group='UNCHANGED$', init_model='UNCHANGED$',  
                init_score='UNCHANGED$', sample_weight='UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see [sklearn.set_config\(\)](#)). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **callbacks** (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `callbacks` parameter in `fit`.
- **categorical_feature** (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `categorical_feature` parameter in `fit`.
- **eval_class_weight** (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_class_weight` parameter in `fit`.
- **eval_group** (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_group` parameter in `fit`.

- **eval_init_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_init_score parameter in fit.
- **eval_metric** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_metric parameter in fit.
- **eval_names** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_names parameter in fit.
- **eval_sample_weight** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_sample_weight parameter in fit.
- **eval_set** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_set parameter in fit.
- **feature_name** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for feature_name parameter in fit.
- **group** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for group parameter in fit.
- **init_model** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for init_model parameter in fit.
- **init_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for init_score parameter in fit.
- **sample_weight** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for sample_weight parameter in fit.

Returns

self – The updated object.

Return type

object

set_params(params)**

Set the parameters of this estimator.

Parameters

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

Returns

self – Returns self.

Return type

object

set_predict_request(*, num_iteration='\$UNCHANGED\$', pred_contrib='\$UNCHANGED\$', pred_leaf='\$UNCHANGED\$', raw_score='\$UNCHANGED\$', start_iteration='\$UNCHANGED\$', validate_features='\$UNCHANGED\$')

Configure whether metadata should be requested to be passed to the `predict` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- `num_iteration` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `num_iteration` parameter in `predict`.
- `pred_contrib` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_contrib` parameter in `predict`.
- `pred_leaf` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_leaf` parameter in `predict`.
- `raw_score` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `raw_score` parameter in `predict`.
- `start_iteration` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `start_iteration` parameter in `predict`.
- `validate_features` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `validate_features` parameter in `predict`.

Returns

`self` – The updated object.

Return type

object

9.3.2 `lightgbm.LGBMClassifier`

```
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=None, importance_type='split', **kwargs)
```

Bases: `ClassifierMixin, LGBMModel`

LightGBM classifier.

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

Construct a gradient boosting model.

Parameters

- **boosting_type** (`str, optional (default='gbdt')`) – ‘gbdt’, traditional Gradient Boosting Decision Tree. ‘dart’, Dropouts meet Multiple Additive Regression Trees. ‘rf’, Random Forest.
- **num_leaves** (`int, optional (default=31)`) – Maximum tree leaves for base learners.
- **max_depth** (`int, optional (default=-1)`) – Maximum tree depth for base learners, $<= 0$ means no limit. If setting this to a positive value, consider also changing `num_leaves` to $<= 2^{\max_depth}$.
- **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** (`str, callable or None, optional (default=None)`) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). Default: ‘regression’ for `LGBMRegressor`, ‘binary’ or ‘multiclass’ for `LGBMClassifier`, ‘lambdarank’ for `LGBMRanker`.
- **class_weight** (`dict, 'balanced' or None, optional (default=None)`) – Weights associated with classes in the form `{class_label: weight}`. Use this parameter only for multi-class classification task; for binary classification task you may use `is_unbalance` or `scale_pos_weight` parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (<https://scikit-learn.org/stable/modules/calibration.html>) of your model. The ‘balanced’ mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`. If `None`, all classes are supposed to have weight one. Note, that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.

- **min_split_gain** (*float, optional (default=0.)*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*float, optional (default=1e-3)*) – Minimum sum of instance weight (Hessian) needed in a child (leaf).
- **min_child_samples** (*int, optional (default=20)*) – Minimum number of data needed in a child (leaf).
- **subsample** (*float, optional (default=1.)*) – Subsample ratio of the training instance.
- **subsample_freq** (*int, optional (default=0)*) – Frequency of subsample, ≤ 0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, RandomState object or None, optional (default=None)*) – Random number seed. If int, this number is used to seed the C++ code. If RandomState or Generator object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.
- **n_jobs** (*int or None, optional (default=None)*) – Number of parallel threads to use for training (can be changed at prediction time by passing it as an extra keyword argument).

For better performance, it is recommended to set this to the number of physical cores in the CPU.

Negative integers are interpreted as following joblib's formula ($n_cpus + 1 + n_jobs$), just like scikit-learn (so e.g. -1 means using all threads). A value of zero corresponds the default number of threads configured for OpenMP in the system. A value of `None` (the default) corresponds to using the number of physical cores in the system (its correct detection requires either the `joblib` or the `psutil` util libraries to be installed).

Changed in version 4.0.0.

- **importance_type** (*str, optional (default='split')*) – The type of feature importance to be filled into `feature_importances_`. If 'split', result contains numbers of times the feature is used in a model. If 'gain', result contains total gains of splits which use the feature.
- ****kwargs** – Other parameters for the model. Check <http://lightgbm.readthedocs.io/en/latest/Parameters.html> for more parameters.

 **Warning**

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

Note

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

`y_true`

[numpy 1-D array of shape = [n_samples]] The target values.

`y_pred`

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

`weight`

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

`group`

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

`grad`

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `y_pred` for each sample point.

`hess`

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of `y_pred` for each sample point.

For multi-class task, `y_pred` is a numpy 2-D array of shape = [n_samples, n_classes], and `grad` and `hess` should be returned in the same format.

Methods

<code>__init__(*[..., boosting_type, num_leaves, ...])</code>	Construct a gradient boosting model.
<code>fit(X, y[, sample_weight, init_score, ...])</code>	Build a gradient boosting model from the training set (X, y).
<code>get_metadata_routing()</code>	Get metadata routing of this object.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, raw_score, start_iteration, ...])</code>	Return the predicted value for each sample.
<code>predict_proba(X[, raw_score, ...])</code>	Return the predicted probability for each class for each sample.
<code>score(X, y[, sample_weight])</code>	Return <code>accuracy</code> on provided data and labels.
<code>set_fit_request(*[, callbacks, ...])</code>	Configure whether metadata should be requested to be passed to the <code>fit</code> method.
<code>set_params(**params)</code>	Set the parameters of this estimator.

continues on next page

Table 11 – continued from previous page

<code>set_predict_proba_request(*[, ...])</code>	Configure whether metadata should be requested to be passed to the <code>predict_proba</code> method.
<code>set_predict_request(*[, num_iteration, ...])</code>	Configure whether metadata should be requested to be passed to the <code>predict</code> method.
<code>set_score_request(*[, sample_weight])</code>	Configure whether metadata should be requested to be passed to the <code>score</code> method.

Attributes

<code>best_iteration_</code>	The best iteration of fitted model if <code>early_stopping()</code> callback has been specified.
<code>best_score_</code>	The best score of fitted model.
<code>booster_</code>	The underlying Booster of this model.
<code>classes_</code>	The class label array.
<code>evals_result_</code>	The evaluation results if validation sets have been specified.
<code>feature_importances_</code>	The feature importances (the higher, the more important).
<code>feature_name_</code>	The names of features.
<code>feature_names_in_</code>	scikit-learn compatible version of <code>feature_name_</code> .
<code>n_classes_</code>	The number of classes.
<code>n_estimators_</code>	True number of boosting iterations performed.
<code>n_features_</code>	The number of features of fitted model.
<code>n_features_in_</code>	The number of features of fitted model.
<code>n_iter_</code>	True number of boosting iterations performed.
<code>objective_</code>	The concrete objective used while fitting this model.

`property best_iteration_`

The best iteration of fitted model if `early_stopping()` callback has been specified.

Type
int

`property best_score_`

The best score of fitted model.

Type
dict

`property booster_`

The underlying Booster of this model.

Type
Booster

`property classes_`

The class label array.

Type
array of shape = [n_classes]

`property evals_result_`

The evaluation results if validation sets have been specified.

Type

dict

property feature_importances_

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

Note

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

Type

array of shape = [n_features]

property feature_name_

The names of features.

Note

If input does not contain feature names, they will be added during fitting in the format `Column_0`, `Column_1`, ..., `Column_N`.

Type

list of shape = [n_features]

property feature_names_in_

scikit-learn compatible version of `.feature_name_`.

Added in version 4.5.0.

Type

array of shape = [n_features]

```
fit(X, y, sample_weight=None, init_score=None, eval_set=None, eval_names=None,
     eval_sample_weight=None, eval_class_weight=None, eval_init_score=None, eval_metric=None,
     feature_name='auto', categorical_feature='auto', callbacks=None, init_model=None)
```

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

Parameters

- **X** (`numpy array`, `pandas DataFrame`, `pyarrow Table`, `scipy.sparse`, `list of lists of int or float of shape = [n_samples, n_features]`) – Input feature matrix.
- **y** (`numpy array`, `pandas DataFrame`, `pandas Series`, `list of int or float`, `pyarrow Array`, `pyarrow ChunkedArray of shape = [n_samples]`) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (`numpy array`, `pandas Series`, `list of int or float`, `pyarrow Array`, `pyarrow ChunkedArray of shape = [n_samples]` or `None`, `optional (default=None)`) – Weights of training data. Weights should be non-negative.

- **init_score** (numpy array, pandas DataFrame, pandas Series, list of int or float, list of lists, pyarrow Array, pyarrow ChunkedArray, pyarrow Table of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task) or shape = [n_samples, n_classes] (for multi-class task) or None, optional (default=None)) – Init score of training data.
- **eval_set** (list or None, optional (default=None)) – A list of (X, y) tuple pairs to use as validation sets.
- **eval_names** (list of str, or None, optional (default=None)) – Names of eval_set.
- **eval_sample_weight** (list of array (same types as sample_weight supports), or None, optional (default=None)) – Weights of eval data. Weights should be non-negative.
- **eval_class_weight** (list or None, optional (default=None)) – Class weights of eval data.
- **eval_init_score** (list of array (same types as init_score supports), or None, optional (default=None)) – Init score of eval data.
- **eval_metric** (str, callable, list or None, optional (default=None)) – If str, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.
- **feature_name** (list of str, or ‘auto’, optional (default='auto')) – Feature names. If ‘auto’ and data is pandas DataFrame, data columns names are used.
- **categorical_feature** (list of str or int, or ‘auto’, optional (default='auto')) – Categorical features. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify feature_name as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.
- **callbacks** (list of callable, or None, optional (default=None)) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.
- **init_model** (str, `pathlib.Path`, `Booster`, `LGBMModel` or None, optional (default=None)) – Filename of LightGBM model, Booster instance or LGBMModel instance used for continue training.

Returns

`self` – Returns self.

Return type

`LGBMClassifier`

Note

Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns `(eval_name, eval_result, is_higher_better)` or list of `(eval_name, eval_result, is_higher_better)`:

`y_true`

[numpy 1-D array of shape = [n_samples]] The target values.

`y_pred`

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. In case of custom objective, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

`weight`

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

`group`

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

`eval_name`

[str] The name of evaluation function (without whitespace).

`eval_result`

[float] The eval result.

`is_higher_better`

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

`get_metadata_routing()`

Get metadata routing of this object.

Please check [User Guide](#) on how the routing mechanism works.

Returns

`routing` – A `MetadataRequest` encapsulating routing information.

Return type

`MetadataRequest`

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

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

Returns

`params` – Parameter names mapped to their values.

Return type

`dict`

property n_classes_

The number of classes.

Type

int

property n_estimators_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property n_features_

The number of features of fitted model.

Type

int

property n_features_in_

The number of features of fitted model.

Type

int

property n_iter_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property objective_

The concrete objective used while fitting this model.

Type

str or callable

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

Return the predicted value for each sample.

Parameters

- **X** (`numpy array, pandas DataFrame, scipy.sparse, list of lists of int or float of shape = [n_samples, n_features]`) – Input features matrix.
- **raw_score** (`bool, optional (default=False)`) – Whether to predict raw scores.
- **start_iteration** (`int, optional (default=0)`) – Start index of the iteration to predict. If ≤ 0 , starts from the first iteration.

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

 **Note**

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

- **validate_features**(*bool, optional (default=False)*) – If True, ensure that the features used to predict match the ones used to train. Used only if data is pandas DataFrame.
- ****kwargs** – Other parameters for the prediction.

Returns

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

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

Return the predicted probability for each class for each sample.

Parameters

- **X** (*numpy array, pandas DataFrame, scipy.sparse, list of lists of int or float of shape = [n_samples, n_features]*) – Input features matrix.
- **raw_score** (*bool, optional (default=False)*) – Whether to predict raw scores.
- **start_iteration**(*int, optional (default=0)*) – Start index of the iteration to predict. If ≤ 0 , starts from the first iteration.
- **num_iteration**(*int or None, optional (default=None)*) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration ≤ 0 , the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If ≤ 0 , all iterations from start_iteration are used (no limits).
- **pred_leaf**(*bool, optional (default=False)*) – Whether to predict leaf index.
- **pred_contrib**(*bool, optional (default=False)*) – Whether to predict feature contributions.

Note

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

- **`validate_features`** (`bool, optional (default=False)`) – If True, ensure that the features used to predict match the ones used to train. Used only if data is pandas DataFrame.
- **`**kwargs`** – Other parameters for the prediction.

Returns

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

`score`(*X*, *y*, `sample_weight=None`)

Return `accuracy` on provided data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **`X`** (`array-like of shape (n_samples, n_features)`) – Test samples.
- **`y`** (`array-like of shape (n_samples,) or (n_samples, n_outputs)`) – True labels for *X*.
- **`sample_weight`** (`array-like of shape (n_samples,), default=None`) – Sample weights.

Returns

`score` – Mean accuracy of `self.predict(X)` w.r.t. *y*.

Return type

float

`set_fit_request`(**callbacks*=\$UNCHANGED\$, *categorical_feature*=\$UNCHANGED\$,
eval_class_weight=\$UNCHANGED\$, *eval_init_score*=\$UNCHANGED\$,
eval_metric=\$UNCHANGED\$, *eval_names*=\$UNCHANGED\$,
eval_sample_weight=\$UNCHANGED\$, *eval_set*=\$UNCHANGED\$,
feature_name=\$UNCHANGED\$, *init_model*=\$UNCHANGED\$,
init_score=\$UNCHANGED\$, *sample_weight*=\$UNCHANGED\$)

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to `fit`.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **callbacks** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for callbacks parameter in `fit`.
- **categorical_feature** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for categorical_feature parameter in `fit`.
- **eval_class_weight** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for eval_class_weight parameter in `fit`.
- **eval_init_score** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for eval_init_score parameter in `fit`.
- **eval_metric** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for eval_metric parameter in `fit`.
- **eval_names** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for eval_names parameter in `fit`.
- **eval_sample_weight** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for eval_sample_weight parameter in `fit`.
- **eval_set** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for eval_set parameter in `fit`.
- **feature_name** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for feature_name parameter in `fit`.
- **init_model** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for init_model parameter in `fit`.
- **init_score** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for init_score parameter in `fit`.

- `sample_weight` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `sample_weight` parameter in `fit`.

Returns

`self` – The updated object.

Return type

object

set_params(params)**

Set the parameters of this estimator.

Parameters

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

Returns

`self` – Returns self.

Return type

object

set_predict_proba_request(*, num_iteration='\$UNCHANGED\$', pred_contrib='\$UNCHANGED\$', pred_leaf='\$UNCHANGED\$', raw_score='\$UNCHANGED\$', start_iteration='\$UNCHANGED\$', validate_features='\$UNCHANGED\$')

Configure whether metadata should be requested to be passed to the `predict_proba` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict_proba` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict_proba`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- `num_iteration` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `num_iteration` parameter in `predict_proba`.
- `pred_contrib` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_contrib` parameter in `predict_proba`.
- `pred_leaf` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_leaf` parameter in `predict_proba`.

- **raw_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for raw_score parameter in predict_proba.
- **start_iteration** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for start_iteration parameter in predict_proba.
- **validate_features** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for validate_features parameter in predict_proba.

Returns

self – The updated object.

Return type

object

set_predict_request(**num_iteration*='\$UNCHANGED\$', *pred_contrib*='\$UNCHANGED\$',
pred_leaf='\$UNCHANGED\$', *raw_score*='\$UNCHANGED\$',
start_iteration='\$UNCHANGED\$', *validate_features*='\$UNCHANGED\$')

Configure whether metadata should be requested to be passed to the predict method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to predict if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to predict.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **num_iteration** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for num_iteration parameter in predict.
- **pred_contrib** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for pred_contrib parameter in predict.
- **pred_leaf** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for pred_leaf parameter in predict.
- **raw_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for raw_score parameter in predict.

- **start_iteration** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for start_iteration parameter in predict.
- **validate_features** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for validate_features parameter in predict.

Returns

self – The updated object.

Return type

object

set_score_request(*, sample_weight='\$UNCHANGED\$')

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see [sklearn.set_config\(\)](#)). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to `score`.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

9.3.3 `lightgbm.LGBMRegressor`

```
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=None, importance_type='split', **kwargs)
```

Bases: `RegressorMixin`, `LGBMModel`

LightGBM regressor.

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

Construct a gradient boosting model.

Parameters

- **boosting_type** (str, optional (default='gbdt')) – ‘gbdt’, traditional Gradient Boosting Decision Tree. ‘dart’, Dropouts meet Multiple Additive Regression Trees. ‘rf’, Random Forest.
- **num_leaves** (int, optional (default=31)) – Maximum tree leaves for base learners.
- **max_depth** (int, optional (default=-1)) – Maximum tree depth for base learners, $<= 0$ means no limit. If setting this to a positive value, consider also changing `num_leaves` to $<= 2^{\max_depth}$.
- **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**(str, callable or None, optional (default=None)) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). Default: ‘regression’ for `LGBMRegressor`, ‘binary’ or ‘multiclass’ for `LGBMClassifier`, ‘lambdaRank’ for `LGBMRanker`.
- **class_weight** (dict, ‘balanced’ or None, optional (default=None)) – Weights associated with classes in the form `{class_label: weight}`. Use this parameter only for multi-class classification task; for binary classification task you may use `is_unbalance` or `scale_pos_weight` parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (<https://scikit-learn.org/stable/modules/calibration.html>) of your model. The ‘balanced’ mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`. If `None`, all classes are supposed to have weight one. Note, that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.
- **min_split_gain**(float, optional (default=0.)) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight**(float, optional (default=1e-3)) – Minimum sum of instance weight (Hessian) needed in a child (leaf).
- **min_child_samples**(int, optional (default=20)) – Minimum number of data needed in a child (leaf).
- **subsample**(float, optional (default=1.)) – Subsample ratio of the training instance.

- **subsample_freq** (*int, optional (default=0)*) – Frequency of subsample, $<=0$ means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, RandomState object or None, optional (default=None)*) – Random number seed. If int, this number is used to seed the C++ code. If RandomState or Generator object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.
- **n_jobs** (*int or None, optional (default=None)*) – Number of parallel threads to use for training (can be changed at prediction time by passing it as an extra keyword argument).

For better performance, it is recommended to set this to the number of physical cores in the CPU.

Negative integers are interpreted as following joblib's formula ($n_cpus + 1 + n_jobs$), just like scikit-learn (so e.g. -1 means using all threads). A value of zero corresponds the default number of threads configured for OpenMP in the system. A value of None (the default) corresponds to using the number of physical cores in the system (its correct detection requires either the joblib or the psutil util libraries to be installed).

Changed in version 4.0.0.

- **importance_type** (*str, optional (default='split')*) – The type of feature importance to be filled into `feature_importances_`. If 'split', result contains numbers of times the feature is used in a model. If 'gain', result contains total gains of splits which use the feature.
- ****kwargs** – Other parameters for the model. Check <http://lightgbm.readthedocs.io/en/latest/Parameters.html> for more parameters.

 **Warning**

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

 **Note**

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

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples,

`n_classes]` (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

`weight`

[numpy 1-D array of shape = `[n_samples]`] The weight of samples. Weights should be non-negative.

`group`

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

`grad`

[numpy 1-D array of shape = `[n_samples]` or numpy 2-D array of shape = `[n_samples, n_classes]` (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `y_pred` for each sample point.

`hess`

[numpy 1-D array of shape = `[n_samples]` or numpy 2-D array of shape = `[n_samples, n_classes]` (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of `y_pred` for each sample point.

For multi-class task, `y_pred` is a numpy 2-D array of shape = `[n_samples, n_classes]`, and `grad` and `hess` should be returned in the same format.

Methods

<code>__init__(*[..., boosting_type, num_leaves, ...])</code>	Construct a gradient boosting model.
<code>fit(X, y[, sample_weight, init_score, ...])</code>	Build a gradient boosting model from the training set <code>(X, y)</code> .
<code>get_metadata_routing()</code>	Get metadata routing of this object.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, raw_score, start_iteration, ...])</code>	Return the predicted value for each sample.
<code>score(X, y[, sample_weight])</code>	Return <code>coefficient of determination</code> on test data.
<code>set_fit_request(*[, callbacks, ...])</code>	Configure whether metadata should be requested to be passed to the <code>fit</code> method.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>set_predict_request(*[, num_iteration, ...])</code>	Configure whether metadata should be requested to be passed to the <code>predict</code> method.
<code>set_score_request(*[, sample_weight])</code>	Configure whether metadata should be requested to be passed to the <code>score</code> method.

Attributes

<code>best_iteration_</code>	The best iteration of fitted model if <code>early_stopping()</code> callback has been specified.
<code>best_score_</code>	The best score of fitted model.
<code>booster_</code>	The underlying Booster of this model.

continues on next page

Table 14 – continued from previous page

<code>evals_result_</code>	The evaluation results if validation sets have been specified.
<code>feature_importances_</code>	The feature importances (the higher, the more important).
<code>feature_name_</code>	The names of features.
<code>feature_names_in_</code>	scikit-learn compatible version of <code>feature_name_</code> .
<code>n_estimators_</code>	True number of boosting iterations performed.
<code>n_features_</code>	The number of features of fitted model.
<code>n_features_in_</code>	The number of features of fitted model.
<code>n_iter_</code>	True number of boosting iterations performed.
<code>objective_</code>	The concrete objective used while fitting this model.

property best_iteration_

The best iteration of fitted model if `early_stopping()` callback has been specified.

Type
int

property best_score_

The best score of fitted model.

Type
dict

property booster_

The underlying Booster of this model.

Type
Booster

property evals_result_

The evaluation results if validation sets have been specified.

Type
dict

property feature_importances_

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

Note

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

Type
array of shape = [n_features]

property feature_name_

The names of features.

Note

If input does not contain feature names, they will be added during fitting in the format `Column_0`, `Column_1`, ..., `Column_N`.

Type

list of shape = [n_features]

property `feature_names_in_`

scikit-learn compatible version of `.feature_name_`.

Added in version 4.5.0.

Type

array of shape = [n_features]

fit(*X*, *y*, `sample_weight=None`, `init_score=None`, `eval_set=None`, `eval_names=None`, `eval_sample_weight=None`, `eval_init_score=None`, `eval_metric=None`, `feature_name='auto'`, `categorical_feature='auto'`, `callbacks=None`, `init_model=None`)

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

Parameters

- **`X`** (*numpy array*, *pandas DataFrame*, *pyarrow Table*, *scipy.sparse*, *list of lists of int or float of shape = [n_samples, n_features]*) – Input feature matrix.
- **`y`** (*numpy array*, *pandas DataFrame*, *pandas Series*, *list of int or float*, *pyarrow Array*, *pyarrow ChunkedArray of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **`sample_weight`** (*numpy array*, *pandas Series*, *list of int or float*, *pyarrow Array*, *pyarrow ChunkedArray of shape = [n_samples]* or *None*, *optional (default=None)*) – Weights of training data. Weights should be non-negative.
- **`init_score`** (*numpy array*, *pandas DataFrame*, *pandas Series*, *list of int or float*, *list of lists*, *pyarrow Array*, *pyarrow ChunkedArray*, *pyarrow Table of shape = [n_samples]* or *shape = [n_samples * n_classes]* (for multi-class task) or *shape = [n_samples, n_classes]* (for multi-class task) or *None*, *optional (default=None)*) – Init score of training data.
- **`eval_set`** (*list or None*, *optional (default=None)*) – A list of (*X*, *y*) tuple pairs to use as validation sets.
- **`eval_names`** (*list of str*, or *None*, *optional (default=None)*) – Names of eval_set.
- **`eval_sample_weight`** (*list of array* (same types as `sample_weight` supports), or *None*, *optional (default=None)*) – Weights of eval data. Weights should be non-negative.
- **`eval_init_score`** (*list of array* (same types as `init_score` supports), or *None*, *optional (default=None)*) – Init score of eval data.

- **eval_metric** *(str, callable, list or None, optional (default=None))* – If str, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.
- **feature_name** *(list of str, or ‘auto’, optional (default='auto'))* – Feature names. If ‘auto’ and data is pandas DataFrame, data columns names are used.
- **categorical_feature** *(list of str or int, or ‘auto’, optional (default='auto'))* – Categorical features. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify **feature_name** as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.
- **callbacks** *(list of callable, or None, optional (default=None))* – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.
- **init_model** *(str, pathlib.Path, Booster, LGBMModel or None, optional (default=None))* – Filename of LightGBM model, Booster instance or LGBMModel instance used for continue training.

Returns

self – Returns self.

Return type

LGBMRegressor

Note

Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns `(eval_name, eval_result, is_higher_better)` or list of `(eval_name, eval_result, is_higher_better)`:

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. In case of custom **objective**, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

weight

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

group

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with

group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

eval_name

[str] The name of evaluation function (without whitespace).

eval_result

[float] The eval result.

is_higher_better

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

get_metadata_routing()

Get metadata routing of this object.

Please check [User Guide](#) on how the routing mechanism works.

Returns

`routing` – A `MetadataRequest` encapsulating routing information.

Return type

`MetadataRequest`

get_params(deep=True)

Get parameters for this estimator.

Parameters

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

Returns

`params` – Parameter names mapped to their values.

Return type

`dict`

property n_estimators_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

`int`

property n_features_

The number of features of fitted model.

Type

`int`

property n_features_in_

The number of features of fitted model.

Type

`int`

property n_iter_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property objective_

The concrete objective used while fitting this model.

Type

str or callable

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

Return the predicted value for each sample.

Parameters

- **X** (*numpy array*, *pandas DataFrame*, *scipy.sparse*, *list of lists of int* or *float* of *shape = [n_samples, n_features]*) – Input features matrix.
- **raw_score** (*bool*, *optional (default=False)*) – Whether to predict raw scores.
- **start_iteration** (*int*, *optional (default=0)*) – Start index of the iteration to predict. If $<= 0$, starts from the first iteration.
- **num_iteration** (*int or None*, *optional (default=None)*) – Total number of iterations used in the prediction. If *None*, if the best iteration exists and `start_iteration` $<= 0$, the best iteration is used; otherwise, all iterations from `start_iteration` are used (no limits). If $<= 0$, all iterations from `start_iteration` are used (no limits).
- **pred_leaf** (*bool*, *optional (default=False)*) – Whether to predict leaf index.
- **pred_contrib** (*bool*, *optional (default=False)*) – Whether to predict feature contributions.

Note

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

- **validate_features** (*bool*, *optional (default=False)*) – If *True*, ensure that the features used to predict match the ones used to train. Used only if data is *pandas DataFrame*.
- ****kwargs** – Other parameters for the prediction.

Returns

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

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

`score(X, y, sample_weight=None)`

Return coefficient of determination on test data.

The coefficient of determination, R^2 , is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares $((y_{true} - y_{pred})^{** 2}) . sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^{** 2}) . sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y , disregarding the input features, would get a R^2 score of 0.0.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for X .
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns

`score` – R^2 of `self.predict(X)` w.r.t. y .

Return type

float

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

```
set_fit_request(*, callbacks='$UNCHANGED$', categorical_feature='$UNCHANGED$',
                eval_init_score='$UNCHANGED$', eval_metric='$UNCHANGED$',
                eval_names='$UNCHANGED$', eval_sample_weight='$UNCHANGED$',
                eval_set='$UNCHANGED$', feature_name='$UNCHANGED$',
                init_model='$UNCHANGED$', init_score='$UNCHANGED$',
                sample_weight='$UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.

- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- `callbacks` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `callbacks` parameter in `fit`.
- `categorical_feature` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `categorical_feature` parameter in `fit`.
- `eval_init_score` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_init_score` parameter in `fit`.
- `eval_metric` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_metric` parameter in `fit`.
- `eval_names` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_names` parameter in `fit`.
- `eval_sample_weight` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_sample_weight` parameter in `fit`.
- `eval_set` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_set` parameter in `fit`.
- `feature_name` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `feature_name` parameter in `fit`.
- `init_model` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `init_model` parameter in `fit`.
- `init_score` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `init_score` parameter in `fit`.
- `sample_weight` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `sample_weight` parameter in `fit`.

Returns

`self` – The updated object.

Return type

object

`set_params(**params)`

Set the parameters of this estimator.

Parameters

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

Returns

self – Returns self.

Return type

object

```
set_predict_request(*, num_iteration='$UNCHANGED$', pred_contrib='$UNCHANGED$',
                    pred_leaf='$UNCHANGED$', raw_score='$UNCHANGED$',
                    start_iteration='$UNCHANGED$', validate_features='$UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the `predict` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- **True**: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- **False**: metadata is not requested and the meta-estimator will not pass it to `predict`.
- **None**: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- **str**: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **num_iteration** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `num_iteration` parameter in `predict`.
- **pred_contrib** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_contrib` parameter in `predict`.
- **pred_leaf** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_leaf` parameter in `predict`.
- **raw_score** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `raw_score` parameter in `predict`.
- **start_iteration** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `start_iteration` parameter in `predict`.
- **validate_features** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `validate_features` parameter in `predict`.

Returns

self – The updated object.

Return type
object

set_score_request(*, *sample_weight*='\$UNCHANGED\$')

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

`sample_weight` (str, `True`, `False`, or `None`, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `sample_weight` parameter in `score`.

Returns

`self` – The updated object.

Return type

object

9.3.4 `lightgbm.LGBMRanker`

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

Bases: `LGBMModel`

LightGBM ranker.

 **Warning**

scikit-learn doesn't support ranking applications yet, therefore this class is not really compatible with the sklearn ecosystem. Please use this class mainly for training and applying ranking models in common sklearnnish way.

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

Construct a gradient boosting model.

Parameters

- **boosting_type** (*str, optional (default='gbdt')*) – ‘gbdt’, traditional Gradient Boosting Decision Tree. ‘dart’, Dropouts meet Multiple Additive Regression Trees. ‘rf’, Random Forest.
- **num_leaves** (*int, optional (default=31)*) – Maximum tree leaves for base learners.
- **max_depth** (*int, optional (default=-1)*) – Maximum tree depth for base learners, $<= 0$ means no limit. If setting this to a positive value, consider also changing `num_leaves` to $<= 2^{\max_depth}$.
- **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** (*str, callable or None, optional (default=None)*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). Default: ‘regression’ for `LGBMRegressor`, ‘binary’ or ‘multiclass’ for `LGBMClassifier`, ‘lambdaRank’ for `LGBMRanker`.
- **class_weight** (*dict, ‘balanced’ or None, optional (default=None)*) – Weights associated with classes in the form `{class_label: weight}`. Use this parameter only for multi-class classification task; for binary classification task you may use `is_unbalance` or `scale_pos_weight` parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (<https://scikit-learn.org/stable/modules/calibration.html>) of your model. The ‘balanced’ mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`. If `None`, all classes are supposed to have weight one. Note, that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.
- **min_split_gain** (*float, optional (default=0.)*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*float, optional (default=1e-3)*) – Minimum sum of instance weight (Hessian) needed in a child (leaf).
- **min_child_samples** (*int, optional (default=20)*) – Minimum number of data needed in a child (leaf).
- **subsample** (*float, optional (default=1.)*) – Subsample ratio of the training instance.

- **subsample_freq** (*int, optional (default=0)*) – Frequency of subsample, $<=0$ means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, RandomState object or None, optional (default=None)*) – Random number seed. If int, this number is used to seed the C++ code. If RandomState or Generator object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.
- **n_jobs** (*int or None, optional (default=None)*) – Number of parallel threads to use for training (can be changed at prediction time by passing it as an extra keyword argument).

For better performance, it is recommended to set this to the number of physical cores in the CPU.

Negative integers are interpreted as following joblib's formula ($n_cpus + 1 + n_jobs$), just like scikit-learn (so e.g. -1 means using all threads). A value of zero corresponds the default number of threads configured for OpenMP in the system. A value of None (the default) corresponds to using the number of physical cores in the system (its correct detection requires either the joblib or the psutil util libraries to be installed).

Changed in version 4.0.0.

- **importance_type** (*str, optional (default='split')*) – The type of feature importance to be filled into `feature_importances_`. If 'split', result contains numbers of times the feature is used in a model. If 'gain', result contains total gains of splits which use the feature.
- ****kwargs** – Other parameters for the model. Check <http://lightgbm.readthedocs.io/en/latest/Parameters.html> for more parameters.

 **Warning**

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

 **Note**

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

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples,

`n_classes]` (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

`weight`

[numpy 1-D array of shape = `[n_samples]`] The weight of samples. Weights should be non-negative.

`group`

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

`grad`

[numpy 1-D array of shape = `[n_samples]` or numpy 2-D array of shape = `[n_samples, n_classes]` (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `y_pred` for each sample point.

`hess`

[numpy 1-D array of shape = `[n_samples]` or numpy 2-D array of shape = `[n_samples, n_classes]` (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of `y_pred` for each sample point.

For multi-class task, `y_pred` is a numpy 2-D array of shape = `[n_samples, n_classes]`, and `grad` and `hess` should be returned in the same format.

Methods

<code>__init__(*, boosting_type, num_leaves, ...)</code>	Construct a gradient boosting model.
<code>fit(X, y[, sample_weight, init_score, ...])</code>	Build a gradient boosting model from the training set (X, y) .
<code>get_metadata_routing()</code>	Get metadata routing of this object.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, raw_score, start_iteration, ...])</code>	Return the predicted value for each sample.
<code>set_fit_request(*[, callbacks, ...])</code>	Configure whether metadata should be requested to be passed to the <code>fit</code> method.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>set_predict_request(*[, num_iteration, ...])</code>	Configure whether metadata should be requested to be passed to the <code>predict</code> method.

Attributes

<code>best_iteration_</code>	The best iteration of fitted model if <code>early_stopping()</code> callback has been specified.
<code>best_score_</code>	The best score of fitted model.
<code>booster_</code>	The underlying Booster of this model.
<code>evals_result_</code>	The evaluation results if validation sets have been specified.
<code>feature_importances_</code>	The feature importances (the higher, the more important).

continues on next page

Table 16 – continued from previous page

<code>feature_name_</code>	The names of features.
<code>feature_names_in_</code>	scikit-learn compatible version of <code>feature_name_</code> .
<code>n_estimators_</code>	True number of boosting iterations performed.
<code>n_features_</code>	The number of features of fitted model.
<code>n_features_in_</code>	The number of features of fitted model.
<code>n_iter_</code>	True number of boosting iterations performed.
<code>objective_</code>	The concrete objective used while fitting this model.

property best_iteration_

The best iteration of fitted model if `early_stopping()` callback has been specified.

Type

`int`

property best_score_

The best score of fitted model.

Type

`dict`

property booster_

The underlying Booster of this model.

Type

`Booster`

property evals_result_

The evaluation results if validation sets have been specified.

Type

`dict`

property feature_importances_

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

Note

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

Type

array of shape = `[n_features]`

property feature_name_

The names of features.

Note

If input does not contain feature names, they will be added during fitting in the format `Column_0`, `Column_1`, ..., `Column_N`.

Type
list of shape = [n_features]

property feature_names_in_
scikit-learn compatible version of .feature_name_.
Added in version 4.5.0.

Type
array of shape = [n_features]

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, 2, 3, 4, 5), feature_name='auto', categorical_feature='auto', callbacks=None, init_model=None)
Build a gradient boosting model from the training set (X, y).

Parameters

- **X** (numpy array, pandas DataFrame, pyarrow Table, scipy.sparse, list of lists of int or float of shape = [n_samples, n_features]) – Input feature matrix.
- **y** (numpy array, pandas DataFrame, pandas Series, list of int or float, pyarrow Array, pyarrow ChunkedArray of shape = [n_samples]) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (numpy array, pandas Series, list of int or float, pyarrow Array, pyarrow ChunkedArray of shape = [n_samples] or None, optional (default=None)) – Weights of training data. Weights should be non-negative.
- **init_score** (numpy array, pandas DataFrame, pandas Series, list of int or float, list of lists, pyarrow Array, pyarrow ChunkedArray, pyarrow Table of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task) or shape = [n_samples, n_classes] (for multi-class task) or None, optional (default=None)) – Init score of training data.
- **group** (numpy array, pandas Series, pyarrow Array, pyarrow ChunkedArray, list of int or float, or None, optional (default=None)) – Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **eval_set** (list or None, optional (default=None)) – A list of (X, y) tuple pairs to use as validation sets.
- **eval_names** (list of str, or None, optional (default=None)) – Names of eval_set.
- **eval_sample_weight** (list of array (same types as sample_weight supports), or None, optional (default=None)) – Weights of eval data. Weights should be non-negative.
- **eval_init_score** (list of array (same types as init_score supports), or None, optional (default=None)) – Init score of eval data.

- **eval_group** (list of array (same types as group supports), or None, optional (default=None)) – Group data of eval data.
- **eval_metric** (*str, callable, list or None, optional (default=None)*) – If str, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.
- **eval_at** (*list or tuple of int, optional (default=(1, 2, 3, 4, 5))*) – The evaluation positions of the specified metric.
- **feature_name** (*list of str, or 'auto', optional (default='auto')*) – Feature names. If ‘auto’ and data is pandas DataFrame, data columns names are used.
- **categorical_feature** (*list of str or int, or 'auto', optional (default='auto')*) – Categorical features. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify feature_name as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.
- **callbacks** (*list of callable, or None, optional (default=None)*) – List of callback functions that are applied at each iteration. See Callbacks in Python API for more information.
- **init_model** (*str, pathlib.Path, Booster, LGBMModel or None, optional (default=None)*) – Filename of LightGBM model, Booster instance or LGBMModel instance used for continue training.

Returns

self – Returns self.

Return type

LGBMRanker

 **Note**

Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns (`eval_name`, `eval_result`, `is_higher_better`) or list of (`eval_name`, `eval_result`, `is_higher_better`):

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. In case of custom objective, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

weight

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

group

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

eval_name

[str] The name of evaluation function (without whitespace).

eval_result

[float] The eval result.

is_higher_better

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

get_metadata_routing()

Get metadata routing of this object.

Please check [User Guide](#) on how the routing mechanism works.

Returns

`routing` – A `MetadataRequest` encapsulating routing information.

Return type

`MetadataRequest`

get_params(deep=True)

Get parameters for this estimator.

Parameters

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

Returns

`params` – Parameter names mapped to their values.

Return type

`dict`

property n_estimators

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

`int`

property n_features

The number of features of fitted model.

Type

`int`

property n_features_in_

The number of features of fitted model.

Type

int

property n_iter_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property objective_

The concrete objective used while fitting this model.

Type

str or callable

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

Return the predicted value for each sample.

Parameters

- **X** (*numpy array*, *pandas DataFrame*, *scipy.sparse*, *list of lists of int or float* of shape $= [n_samples, n_features]$) – Input features matrix.
- **raw_score** (*bool*, *optional (default=False)*) – Whether to predict raw scores.
- **start_iteration** (*int, optional (default=0)*) – Start index of the iteration to predict. If ≤ 0 , starts from the first iteration.
- **num_iteration** (*int or None, optional (default=None)*) – Total number of iterations used in the prediction. If *None*, if the best iteration exists and `start_iteration` ≤ 0 , the best iteration is used; otherwise, all iterations from `start_iteration` are used (no limits). If ≤ 0 , all iterations from `start_iteration` are used (no limits).
- **pred_leaf** (*bool, optional (default=False)*) – Whether to predict leaf index.
- **pred_contrib** (*bool, optional (default=False)*) – Whether to predict feature contributions.

Note

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

- **validate_features** (*bool, optional (default=False)*) – If *True*, ensure that the features used to predict match the ones used to train. Used only if data is *pandas DataFrame*.

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

Returns

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

```
set_fit_request(*, callbacks='UNCHANGED$', categorical_feature='UNCHANGED$',  
eval_at='UNCHANGED$', eval_group='UNCHANGED$',  
eval_init_score='UNCHANGED$', eval_metric='UNCHANGED$',  
eval_names='UNCHANGED$', eval_sample_weight='UNCHANGED$',  
eval_set='UNCHANGED$', feature_name='UNCHANGED$',  
group='UNCHANGED$', init_model='UNCHANGED$',  
init_score='UNCHANGED$', sample_weight='UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **callbacks** (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `callbacks` parameter in `fit`.
- **categorical_feature** (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `categorical_feature` parameter in `fit`.
- **eval_at** (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_at` parameter in `fit`.
- **eval_group** (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_group` parameter in `fit`.

- **eval_init_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_init_score parameter in fit.
- **eval_metric** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_metric parameter in fit.
- **eval_names** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_names parameter in fit.
- **eval_sample_weight** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_sample_weight parameter in fit.
- **eval_set** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_set parameter in fit.
- **feature_name** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for feature_name parameter in fit.
- **group** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for group parameter in fit.
- **init_model** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for init_model parameter in fit.
- **init_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for init_score parameter in fit.
- **sample_weight** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for sample_weight parameter in fit.

Returns

self – The updated object.

Return type

object

set_params(params)**

Set the parameters of this estimator.

Parameters

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

Returns

self – Returns self.

Return type

object

set_predict_request(*, num_iteration='\$UNCHANGED\$', pred_contrib='\$UNCHANGED\$', pred_leaf='\$UNCHANGED\$', raw_score='\$UNCHANGED\$', start_iteration='\$UNCHANGED\$', validate_features='\$UNCHANGED\$')

Configure whether metadata should be requested to be passed to the `predict` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- `num_iteration` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `num_iteration` parameter in `predict`.
- `pred_contrib` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_contrib` parameter in `predict`.
- `pred_leaf` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_leaf` parameter in `predict`.
- `raw_score` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `raw_score` parameter in `predict`.
- `start_iteration` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `start_iteration` parameter in `predict`.
- `validate_features` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `validate_features` parameter in `predict`.

Returns

`self` – The updated object.

Return type

`object`

9.4 Dask API

Added in version 3.2.0.

<code>DaskLGBMClassifier(*[, boosting_type, ...])</code>	Distributed version of <code>lightgbm.LGBMClassifier</code> .
<code>DaskLGBMRegressor(*[, boosting_type, ...])</code>	Distributed version of <code>lightgbm.LGBMRegressor</code> .
<code>DaskLGBMRanker(*[, boosting_type, ...])</code>	Distributed version of <code>lightgbm.LGBMRanker</code> .

9.4.1 `lightgbm.DaskLGBMClassifier`

```
class lightgbm.DaskLGBMClassifier(*, 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=None,
                                  importance_type='split', client=None, **kwargs)
```

Bases: `LGBMClassifier`, `_DaskLGBMModel`

Distributed version of `lightgbm.LGBMClassifier`.

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

Construct a gradient boosting model.

Parameters

- **boosting_type** (`str, optional (default='gbdt')`) – ‘gbdt’, traditional Gradient Boosting Decision Tree. ‘dart’, Dropouts meet Multiple Additive Regression Trees. ‘rf’, Random Forest.
- **num_leaves** (`int, optional (default=31)`) – Maximum tree leaves for base learners.
- **max_depth** (`int, optional (default=-1)`) – Maximum tree depth for base learners, $<= 0$ means no limit. If setting this to a positive value, consider also changing `num_leaves` to $<= 2^{\max_depth}$.
- **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** (`str, callable or None, optional (default=None)`) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). Default: ‘regression’ for `LGBMRegressor`, ‘binary’ or ‘multiclass’ for `LGBMClassifier`, ‘lambdarank’ for `LGBMRanker`.
- **class_weight** (`dict, 'balanced' or None, optional (default=None)`) – Weights associated with classes in the form `{class_label: weight}`. Use this parameter only for multi-class classification task; for binary classification task you may use `is_unbalance` or `scale_pos_weight` parameters. Note, that the usage of

all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (<https://scikit-learn.org/stable/modules/calibration.html>) of your model. The ‘balanced’ mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as $n_samples / (n_classes * np.bincount(y))$. If `None`, all classes are supposed to have weight one. Note, that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.

- **`min_split_gain`** (`float, optional (default=0.)`) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **`min_child_weight`** (`float, optional (default=1e-3)`) – Minimum sum of instance weight (Hessian) needed in a child (leaf).
- **`min_child_samples`** (`int, optional (default=20)`) – Minimum number of data needed in a child (leaf).
- **`subsample`** (`float, optional (default=1.)`) – Subsample ratio of the training instance.
- **`subsample_freq`** (`int, optional (default=0)`) – Frequency of subsample, ≤ 0 means no enable.
- **`colsample_bytree`** (`float, optional (default=1.)`) – Subsample ratio of columns when constructing each tree.
- **`reg_alpha`** (`float, optional (default=0.)`) – L1 regularization term on weights.
- **`reg_lambda`** (`float, optional (default=0.)`) – L2 regularization term on weights.
- **`random_state`** (`int, RandomState object or None, optional (default=None)`) – Random number seed. If `int`, this number is used to seed the C++ code. If `RandomState` or `Generator` object (`numpy`), a random integer is picked based on its state to seed the C++ code. If `None`, default seeds in C++ code are used.
- **`n_jobs`** (`int or None, optional (default=None)`) – Number of parallel threads to use for training (can be changed at prediction time by passing it as an extra keyword argument).

For better performance, it is recommended to set this to the number of physical cores in the CPU.

Negative integers are interpreted as following joblib’s formula ($n_cpus + 1 + n_jobs$), just like scikit-learn (so e.g. `-1` means using all threads). A value of zero corresponds the default number of threads configured for OpenMP in the system. A value of `None` (the default) corresponds to using the number of physical cores in the system (its correct detection requires either the `joblib` or the `psutil` util libraries to be installed).

Changed in version 4.0.0.

- **`importance_type`** (`str, optional (default='split')`) – The type of feature importance to be filled into `feature_importances_`. If ‘split’, result contains numbers of times the feature is used in a model. If ‘gain’, result contains total gains of splits which use the feature.
- **`client`** (`task.distributed.Client or None, optional (default=None)`) – Dask client. If `None`, `distributed.default_client()` will be used at runtime.

The Dask client used by this class will not be saved if the model object is pickled.

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

⚠ Warning

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

ⓘ 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`, `objective(y_true, y_pred, weight) -> grad, hess` or `objective(y_true, y_pred, weight, group) -> grad, hess`:

`y_true`

[numpy 1-D array of shape = [n_samples]] The target values.

`y_pred`

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

`weight`

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

`group`

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

`grad`

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `y_pred` for each sample point.

`hess`

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of `y_pred` for each sample point.

For multi-class task, `y_pred` is a numpy 2-D array of shape = [n_samples, n_classes], and `grad` and `hess` should be returned in the same format.

Methods

<code>__init__</code> (*[, boosting_type, num_leaves, ...])	Construct a gradient boosting model.
<code>fit</code> (X, y[, sample_weight, init_score, ...])	Build a gradient boosting model from the training set (X, y).

continues on next page

Table 18 – continued from previous page

<code>get_metadata_routing()</code>	Get metadata routing of this object.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, raw_score, start_iteration, ...])</code>	Return the predicted value for each sample.
<code>predict_proba(X[, raw_score, ...])</code>	Return the predicted probability for each class for each sample.
<code>score(X, y[, sample_weight])</code>	Return <code>accuracy</code> on provided data and labels.
<code>set_fit_request(*[, eval_class_weight, ...])</code>	Configure whether metadata should be requested to be passed to the <code>fit</code> method.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>set_predict_proba_request(*[, ...])</code>	Configure whether metadata should be requested to be passed to the <code>predict_proba</code> method.
<code>set_predict_request(*[, num_iteration, ...])</code>	Configure whether metadata should be requested to be passed to the <code>predict</code> method.
<code>set_score_request(*[, sample_weight])</code>	Configure whether metadata should be requested to be passed to the <code>score</code> method.
<code>to_local()</code>	Create regular version of <code>lightgbm.LGBMClassifier</code> from the distributed version.

Attributes

<code>best_iteration_</code>	The best iteration of fitted model if <code>early_stopping()</code> callback has been specified.
<code>best_score_</code>	The best score of fitted model.
<code>booster_</code>	The underlying Booster of this model.
<code>classes_</code>	The class label array.
<code>client_</code>	Dask client.
<code>evals_result_</code>	The evaluation results if validation sets have been specified.
<code>feature_importances_</code>	The feature importances (the higher, the more important).
<code>feature_name_</code>	The names of features.
<code>feature_names_in_</code>	scikit-learn compatible version of <code>feature_name_</code> .
<code>n_classes_</code>	The number of classes.
<code>n_estimators_</code>	True number of boosting iterations performed.
<code>n_features_</code>	The number of features of fitted model.
<code>n_features_in_</code>	The number of features of fitted model.
<code>n_iter_</code>	True number of boosting iterations performed.
<code>objective_</code>	The concrete objective used while fitting this model.

`property best_iteration_`

The best iteration of fitted model if `early_stopping()` callback has been specified.

Type
int

`property best_score_`

The best score of fitted model.

Type
dict

property booster_

The underlying Booster of this model.

Type

Booster

property classes_

The class label array.

Type

array of shape = [n_classes]

property client_

Dask client.

This property can be passed in the constructor or updated with `model.set_params(client=client)`.

Type

`dask.distributed.Client`

property evals_result_

The evaluation results if validation sets have been specified.

Type

`dict`

property feature_importances_

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

Note

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

Type

array of shape = [n_features]

property feature_name_

The names of features.

Note

If input does not contain feature names, they will be added during fitting in the format `Column_0`, `Column_1`, ..., `Column_N`.

Type

list of shape = [n_features]

property feature_names_in_

scikit-learn compatible version of `.feature_name_`.

Added in version 4.5.0.

Type

array of shape = [n_features]

```
fit(X, y, sample_weight=None, init_score=None, eval_set=None, eval_names=None,
eval_sample_weight=None, eval_class_weight=None, eval_init_score=None, eval_metric=None,
**kwargs)
```

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

Parameters

- **X** (*Dask Array or Dask DataFrame of shape = [n_samples, n_features]*) – Input feature matrix.
- **y** (*Dask Array, Dask DataFrame or Dask Series of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (*Dask Array or Dask Series of shape = [n_samples] or None, optional (default=None)*) – Weights of training data. Weights should be non-negative.
- **init_score** (*Dask Array or Dask Series of shape = [n_samples] or shape = [n_samples * n_classes] (for multi-class task), or Dask Array or Dask DataFrame of shape = [n_samples, n_classes] (for multi-class task), or None, optional (default=None)*) – Init score of training data.
- **eval_set** (*list or None, optional (default=None)*) – A list of (X, y) tuple pairs to use as validation sets.
- **eval_names** (*list of str, or None, optional (default=None)*) – Names of eval_set.
- **eval_sample_weight** (*list of Dask Array or Dask Series, or None, optional (default=None)*) – Weights of eval data. Weights should be non-negative.
- **eval_class_weight** (*list or None, optional (default=None)*) – Class weights of eval data.
- **eval_init_score** (*list of Dask Array, Dask Series or Dask DataFrame (for multi-class task), or None, optional (default=None)*) – Init score of eval data.
- **eval_metric** (*str, callable, list or None, optional (default=None)*) – If str, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.
- **feature_name** (*list of str, or ‘auto’, optional (default='auto')*) – Feature names. If ‘auto’ and data is pandas DataFrame, data columns names are used.
- **categorical_feature** (*list of str or int, or ‘auto’, optional (default='auto')*) – Categorical features. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify **feature_name** as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained

with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.

- ****kwargs** – Other parameters passed through to `LGBMClassifier.fit()`.

Returns

`self` – Returns self.

Return type

`lightgbm.DaskLGBMClassifier`

Note

Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns (`eval_name, eval_result, is_higher_better`) or list of (`eval_name, eval_result, is_higher_better`):

`y_true`

[numpy 1-D array of shape = [n_samples]] The target values.

`y_pred`

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. In case of custom `objective`, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

`weight`

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

`group`

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

`eval_name`

[str] The name of evaluation function (without whitespace).

`eval_result`

[float] The eval result.

`is_higher_better`

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

`get_metadata_routing()`

Get metadata routing of this object.

Please check [User Guide](#) on how the routing mechanism works.

Returns

`routing` – A `MetadataRequest` encapsulating routing information.

Return type

`MetadataRequest`

`get_params(deep=True)`

Get parameters for this estimator.

Parameters

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

Returns

params – Parameter names mapped to their values.

Return type

dict

property n_classes_

The number of classes.

Type

int

property n_estimators_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property n_features_

The number of features of fitted model.

Type

int

property n_features_in_

The number of features of fitted model.

Type

int

property n_iter_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property objective_

The concrete objective used while fitting this model.

Type

str or callable

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

Return the predicted value for each sample.

Parameters

- **X** (*Dask Array or Dask DataFrame of shape = [n_samples, n_features]*) – Input features matrix.
- **raw_score** (*bool, optional (default=False)*) – Whether to predict raw scores.
- **start_iteration** (*int, optional (default=0)*) – Start index of the iteration to predict. If $<= 0$, starts from the first iteration.
- **num_iteration** (*int or None, optional (default=None)*) – Total number of iterations used in the prediction. If None, if the best iteration exists and start_iteration $<= 0$, the best iteration is used; otherwise, all iterations from start_iteration are used (no limits). If $<= 0$, all iterations from start_iteration are used (no limits).
- **pred_leaf** (*bool, optional (default=False)*) – Whether to predict leaf index.
- **pred_contrib** (*bool, optional (default=False)*) – Whether to predict feature contributions.

Note

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

- **validate_features** (*bool, optional (default=False)*) – If True, ensure that the features used to predict match the ones used to train. Used only if data is pandas DataFrame.
- ****kwargs** – Other parameters for the prediction.

Returns

- **predicted_result** (*Dask Array of shape = [n_samples] or shape = [n_samples, n_classes]*) – The predicted values.
- **X_leaves** (*Dask Array of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes]*) – If `pred_leaf=True`, the predicted leaf of every tree for each sample.
- **X_SHAP_values** (*Dask Array of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes]* or (if multi-class and using sparse inputs) a list of `n_classes` Dask Arrays of shape = [n_samples, n_features + 1]) – If `pred_contrib=True`, the feature contributions for each sample.

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

Return the predicted probability for each class for each sample.

Parameters

- **X** (*Dask Array or Dask DataFrame of shape = [n_samples, n_features]*) – Input features matrix.
- **raw_score** (*bool, optional (default=False)*) – Whether to predict raw scores.
- **start_iteration** (*int, optional (default=0)*) – Start index of the iteration to predict. If $<= 0$, starts from the first iteration.

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

 **Note**

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

- **validate_features** (*bool, optional (default=False)*) – If True, ensure that the features used to predict match the ones used to train. Used only if data is pandas DataFrame.
- ****kwargs** – Other parameters for the prediction.

Returns

- **predicted_probability** (*Dask Array of shape = [n_samples] or shape = [n_samples, n_classes]*) – The predicted values.
- **X_leaves** (*Dask Array of shape = [n_samples, n_trees] or shape = [n_samples, n_trees * n_classes]*) – If pred_leaf=True, the predicted leaf of every tree for each sample.
- **X_SHAP_values** (*Dask Array of shape = [n_samples, n_features + 1] or shape = [n_samples, (n_features + 1) * n_classes]* or (if multi-class and using sparse inputs) a list of n_classes Dask Arrays of shape = [n_samples, n_features + 1]) – If pred_contrib=True, the feature contributions for each sample.

score(*X, y, sample_weight=None*)

Return **accuracy** on provided data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for X.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns

score – Mean accuracy of self.predict(X) w.r.t. y.

Return type

float

```
set_fit_request(*, eval_class_weight='$UNCHANGED$', eval_init_score='$UNCHANGED$',  
                eval_metric='$UNCHANGED$', eval_names='$UNCHANGED$',  
                eval_sample_weight='$UNCHANGED$', eval_set='$UNCHANGED$',  
                init_score='$UNCHANGED$', sample_weight='$UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the `fit` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **`eval_class_weight`** (str, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_class_weight` parameter in `fit`.
- **`eval_init_score`** (str, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_init_score` parameter in `fit`.
- **`eval_metric`** (str, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_metric` parameter in `fit`.
- **`eval_names`** (str, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_names` parameter in `fit`.
- **`eval_sample_weight`** (str, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_sample_weight` parameter in `fit`.
- **`eval_set`** (str, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_set` parameter in `fit`.
- **`init_score`** (str, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `init_score` parameter in `fit`.
- **`sample_weight`** (str, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `sample_weight` parameter in `fit`.

Returns

`self` – The updated object.

Return type
object

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

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

Returns
self – Returns self.

Return type
object

set_predict_proba_request(*, num_iteration='UNCHANGED', pred_contrib='UNCHANGED', pred_leaf='UNCHANGED', raw_score='UNCHANGED', start_iteration='UNCHANGED', validate_features='UNCHANGED')

Configure whether metadata should be requested to be passed to the `predict_proba` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to `predict_proba` if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to `predict_proba`.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- `num_iteration` (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `num_iteration` parameter in `predict_proba`.
- `pred_contrib` (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_contrib` parameter in `predict_proba`.
- `pred_leaf` (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_leaf` parameter in `predict_proba`.
- `raw_score` (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `raw_score` parameter in `predict_proba`.
- `start_iteration` (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `start_iteration` parameter in `predict_proba`.

- **validate_features** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for validate_features parameter in predict_proba.

Returns

self – The updated object.

Return type

object

```
set_predict_request(*, num_iteration='$UNCHANGED$', pred_contrib='$UNCHANGED$',  
                    pred_leaf='$UNCHANGED$', raw_score='$UNCHANGED$',  
                    start_iteration='$UNCHANGED$', validate_features='$UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the predict method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a [meta-estimator](#) and metadata routing is enabled with enable_metadata_routing=True (see [sklearn.set_config\(\)](#)). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to predict if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to predict.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (sklearn.utils.metadata_routing.UNCHANGED) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **num_iteration** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for num_iteration parameter in predict.
- **pred_contrib** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for pred_contrib parameter in predict.
- **pred_leaf** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for pred_leaf parameter in predict.
- **raw_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for raw_score parameter in predict.
- **start_iteration** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for start_iteration parameter in predict.
- **validate_features** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for validate_features parameter in predict.

Returns**self** – The updated object.**Return type**

object

set_score_request(**, sample_weight='UNCHANGED'*)Configure whether metadata should be requested to be passed to the `score` method.Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters**sample_weight** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `sample_weight` parameter in `score`.**Returns****self** – The updated object.**Return type**

object

to_local()Create regular version of `lightgbm.LGBMClassifier` from the distributed version.**Returns****model** – Local underlying model.**Return type**`lightgbm.LGBMClassifier`

9.4.2 `lightgbm.DaskLGBMRegressor`

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

Bases: `LGBMRegressor`, `_DaskLGBMModel`Distributed version of `lightgbm.LGBMRegressor`.

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

Construct a gradient boosting model.

Parameters

- **boosting_type** (*str, optional (default='gbdt')*) – ‘gbdt’, traditional Gradient Boosting Decision Tree. ‘dart’, Dropouts meet Multiple Additive Regression Trees. ‘rf’, Random Forest.
- **num_leaves** (*int, optional (default=31)*) – Maximum tree leaves for base learners.
- **max_depth** (*int, optional (default=-1)*) – Maximum tree depth for base learners, $<= 0$ means no limit. If setting this to a positive value, consider also changing `num_leaves` to $<= 2^{\max_depth}$.
- **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** (*str, callable or None, optional (default=None)*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). Default: ‘regression’ for `LGBMRegressor`, ‘binary’ or ‘multiclass’ for `LGBMClassifier`, ‘lambdarank’ for `LGBMRanker`.
- **class_weight** (*dict, ‘balanced’ or None, optional (default=None)*) – Weights associated with classes in the form `{class_label: weight}`. Use this parameter only for multi-class classification task; for binary classification task you may use `is_unbalance` or `scale_pos_weight` parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (<https://scikit-learn.org/stable/modules/calibration.html>) of your model. The ‘balanced’ mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`. If `None`, all classes are supposed to have weight one. Note, that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.
- **min_split_gain** (*float, optional (default=0.)*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*float, optional (default=1e-3)*) – Minimum sum of instance weight (Hessian) needed in a child (leaf).
- **min_child_samples** (*int, optional (default=20)*) – Minimum number of data needed in a child (leaf).
- **subsample** (*float, optional (default=1.)*) – Subsample ratio of the training instance.

- **subsample_freq** (*int, optional (default=0)*) – Frequency of subsample, $<=0$ means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, RandomState object or None, optional (default=None)*) – Random number seed. If int, this number is used to seed the C++ code. If RandomState or Generator object (numpy), a random integer is picked based on its state to seed the C++ code. If None, default seeds in C++ code are used.
- **n_jobs** (*int or None, optional (default=None)*) – Number of parallel threads to use for training (can be changed at prediction time by passing it as an extra keyword argument).

For better performance, it is recommended to set this to the number of physical cores in the CPU.

Negative integers are interpreted as following joblib's formula ($n_cpus + 1 + n_jobs$), just like scikit-learn (so e.g. -1 means using all threads). A value of zero corresponds the default number of threads configured for OpenMP in the system. A value of None (the default) corresponds to using the number of physical cores in the system (its correct detection requires either the joblib or the psutil util libraries to be installed).

Changed in version 4.0.0.

- **importance_type** (*str, optional (default='split')*) – The type of feature importance to be filled into `feature_importances_`. If 'split', result contains numbers of times the feature is used in a model. If 'gain', result contains total gains of splits which use the feature.
- **client** (*dask.distributed.Client or None, optional (default=None)*) – Dask client. If None, `distributed.default_client()` will be used at runtime. The Dask client used by this class will not be saved if the model object is pickled.
- ****kwargs** – Other parameters for the model. Check <http://lightgbm.readthedocs.io/en/latest/Parameters.html> for more parameters.

⚠ Warning

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

ⓘ 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`, `objective(y_true, y_pred, weight) -> grad, hess` or `objective(y_true, y_pred, weight, group) -> grad, hess`:

y_true	[numpy 1-D array of shape = [n_samples]] The target values.
y_pred	[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.
weight	[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.
group	[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. sum(group) = n_samples. For example, if you have a 100-document dataset with group = [10, 20, 40, 10, 10, 10], that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
grad	[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of y_pred for each sample point.
hess	[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of y_pred for each sample point.

For multi-class task, y_pred is a numpy 2-D array of shape = [n_samples, n_classes], and grad and hess should be returned in the same format.

Methods

<code>__init__(*[boosting_type, num_leaves, ...])</code>	Construct a gradient boosting model.
<code>fit(X, y[, sample_weight, init_score, ...])</code>	Build a gradient boosting model from the training set (X, y).
<code>get_metadata_routing()</code>	Get metadata routing of this object.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, raw_score, start_iteration, ...])</code>	Return the predicted value for each sample.
<code>score(X, y[, sample_weight])</code>	Return coefficient of determination on test data.
<code>set_fit_request(*[, eval_init_score, ...])</code>	Configure whether metadata should be requested to be passed to the fit method.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>set_predict_request(*[, num_iteration, ...])</code>	Configure whether metadata should be requested to be passed to the predict method.
<code>set_score_request(*[, sample_weight])</code>	Configure whether metadata should be requested to be passed to the score method.
<code>to_local()</code>	Create regular version of lightgbm.LGBMRegressor from the distributed version.

Attributes

<code>best_iteration_</code>	The best iteration of fitted model if <code>early_stopping()</code> callback has been specified.
<code>best_score_</code>	The best score of fitted model.
<code>booster_</code>	The underlying Booster of this model.
<code>client_</code>	Dask client.
<code>evals_result_</code>	The evaluation results if validation sets have been specified.
<code>feature_importances_</code>	The feature importances (the higher, the more important).
<code>feature_name_</code>	The names of features.
<code>feature_names_in_</code>	scikit-learn compatible version of <code>feature_name_</code> .
<code>n_estimators_</code>	True number of boosting iterations performed.
<code>n_features_</code>	The number of features of fitted model.
<code>n_features_in_</code>	The number of features of fitted model.
<code>n_iter_</code>	True number of boosting iterations performed.
<code>objective_</code>	The concrete objective used while fitting this model.

`property best_iteration_`

The best iteration of fitted model if `early_stopping()` callback has been specified.

Type

`int`

`property best_score_`

The best score of fitted model.

Type

`dict`

`property booster_`

The underlying Booster of this model.

Type

`Booster`

`property client_`

Dask client.

This property can be passed in the constructor or updated with `model.set_params(client=client)`.

Type

`dask.distributed.Client`

`property evals_result_`

The evaluation results if validation sets have been specified.

Type

`dict`

`property feature_importances_`

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

Note

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

Type

array of shape = [n_features]

property `feature_name_`

The names of features.

Note

If input does not contain feature names, they will be added during fitting in the format `Column_0`, `Column_1`, ..., `Column_N`.

Type

list of shape = [n_features]

property `feature_names_in_`

scikit-learn compatible version of `.feature_name_`.

Added in version 4.5.0.

Type

array of shape = [n_features]

`fit(X, y, sample_weight=None, init_score=None, eval_set=None, eval_names=None, eval_sample_weight=None, eval_init_score=None, eval_metric=None, **kwargs)`

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

Parameters

- **`X`** (*Dask Array or Dask DataFrame of shape = [n_samples, n_features]*) – Input feature matrix.
- **`y`** (*Dask Array, Dask DataFrame or Dask Series of shape = [n_samples]*) – The target values (class labels in classification, real numbers in regression).
- **`sample_weight`** (*Dask Array or Dask Series of shape = [n_samples] or None, optional (default=None)*) – Weights of training data. Weights should be non-negative.
- **`init_score`** (*Dask Array or Dask Series of shape = [n_samples] or None, optional (default=None)*) – Init score of training data.
- **`eval_set`** (*list or None, optional (default=None)*) – A list of (X, y) tuple pairs to use as validation sets.
- **`eval_names`** (*list of str, or None, optional (default=None)*) – Names of eval_set.
- **`eval_sample_weight`** (*list of Dask Array or Dask Series, or None, optional (default=None)*) – Weights of eval data. Weights should be non-negative.

- **eval_init_score** (*list of Dask Array or Dask Series, or None, optional (default=None)*) – Init score of eval data.
- **eval_metric** (*str, callable, list or None, optional (default=None)*) – If str, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.
- **feature_name** (*list of str, or ‘auto’, optional (default='auto')*) – Feature names. If ‘auto’ and data is pandas DataFrame, data columns names are used.
- **categorical_feature** (*list of str or int, or ‘auto’, optional (default='auto')*) – Categorical features. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify feature_name as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32 max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.
- ****kwargs** – Other parameters passed through to LGBMRegressor.fit().

Returns

self – Returns self.

Return type

lightgbm.DaskLGBMRegressor

Note

Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns (`eval_name, eval_result, is_higher_better`) or list of (`eval_name, eval_result, is_higher_better`):

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. In case of custom objective, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

weight

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

group

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

eval_name
[str] The name of evaluation function (without whitespace).
eval_result
[float] The eval result.
is_higher_better
[bool] Is eval result higher better, e.g. AUC is <code>is_higher_better</code> .

get_metadata_routing()

Get metadata routing of this object.

Please check [User Guide](#) on how the routing mechanism works.

Returns

`routing` – A `MetadataRequest` encapsulating routing information.

Return type

`MetadataRequest`

get_params(deep=True)

Get parameters for this estimator.

Parameters

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

Returns

`params` – Parameter names mapped to their values.

Return type

`dict`

property n_estimators_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

`int`

property n_features_

The number of features of fitted model.

Type

`int`

property n_features_in_

The number of features of fitted model.

Type

`int`

property n_iter_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property objective_

The concrete objective used while fitting this model.

Type

str or callable

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

Return the predicted value for each sample.

Parameters

- **X** (*Dask Array or Dask DataFrame of shape = [n_samples, n_features]*) – Input features matrix.
- **raw_score** (*bool, optional (default=False)*) – Whether to predict raw scores.
- **start_iteration** (*int, optional (default=0)*) – Start index of the iteration to predict. If ≤ 0 , starts from the first iteration.
- **num_iteration** (*int or None, optional (default=None)*) – Total number of iterations used in the prediction. If None, if the best iteration exists and *start_iteration* ≤ 0 , the best iteration is used; otherwise, all iterations from *start_iteration* are used (no limits). If ≤ 0 , all iterations from *start_iteration* are used (no limits).
- **pred_leaf** (*bool, optional (default=False)*) – Whether to predict leaf index.
- **pred_contrib** (*bool, optional (default=False)*) – Whether to predict feature contributions.

 **Note**

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

- **validate_features** (*bool, optional (default=False)*) – If True, ensure that the features used to predict match the ones used to train. Used only if data is pandas DataFrame.
- ****kwargs** – Other parameters for the prediction.

Returns

- **predicted_result** (*Dask Array of shape = [n_samples]*) – The predicted values.
- **X_leaves** (*Dask Array of shape = [n_samples, n_trees]*) – If *pred_leaf=True*, the predicted leaf of every tree for each sample.
- **X_SHAP_values** (*Dask Array of shape = [n_samples, n_features + 1]*) – If *pred_contrib=True*, the feature contributions for each sample.

score(*X*, *y*, *sample_weight=None*)

Return coefficient of determination on test data.

The coefficient of determination, R^2 , is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares $((y_{true} - y_{pred})^2).sum()$ and v is the total sum of squares $((y_{true} - y_{true}.mean())^2).sum()$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0.

Parameters

- **X** (array-like of shape (n_samples, n_features)) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.
- **y** (array-like of shape (n_samples,) or (n_samples, n_outputs)) – True values for *X*.
- **sample_weight** (array-like of shape (n_samples,), default=None) – Sample weights.

Returns

score – R^2 of self.predict(*X*) w.r.t. *y*.

Return type

float

Notes

The R^2 score used when calling **score** on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the **score** method of all the multioutput regressors (except for `MultiOutputRegressor`).

```
set_fit_request(*, eval_init_score='$UNCHANGED$', eval_metric='$UNCHANGED$',  
                eval_names='$UNCHANGED$', eval_sample_weight='$UNCHANGED$',  
                eval_set='$UNCHANGED$', init_score='$UNCHANGED$',  
                sample_weight='$UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the **fit** method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a meta-estimator and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- **True**: metadata is requested, and passed to **fit** if provided. The request is ignored if metadata is not provided.
- **False**: metadata is not requested and the meta-estimator will not pass it to **fit**.
- **None**: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- **str**: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **eval_init_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_init_score parameter in fit.
- **eval_metric** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_metric parameter in fit.
- **eval_names** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_names parameter in fit.
- **eval_sample_weight** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_sample_weight parameter in fit.
- **eval_set** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for eval_set parameter in fit.
- **init_score** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for init_score parameter in fit.
- **sample_weight** (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for sample_weight parameter in fit.

Returns

self – The updated object.

Return type

object

set_params(params)**

Set the parameters of this estimator.

Parameters

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

Returns

self – Returns self.

Return type

object

set_predict_request(*, num_iteration='\$UNCHANGED\$', pred_contrib='\$UNCHANGED\$', pred_leaf='\$UNCHANGED\$', raw_score='\$UNCHANGED\$', start_iteration='\$UNCHANGED\$', validate_features='\$UNCHANGED\$')

Configure whether metadata should be requested to be passed to the predict method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to `predict`.

- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- `num_iteration` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `num_iteration` parameter in `predict`.
- `pred_contrib` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_contrib` parameter in `predict`.
- `pred_leaf` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_leaf` parameter in `predict`.
- `raw_score` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `raw_score` parameter in `predict`.
- `start_iteration` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `start_iteration` parameter in `predict`.
- `validate_features` (`str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `validate_features` parameter in `predict`.

Returns

`self` – The updated object.

Return type

object

`set_score_request(*, sample_weight='$UNCHANGED$')`

Configure whether metadata should be requested to be passed to the `score` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

sample_weight (str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED) – Metadata routing for `sample_weight` parameter in `score`.

Returns

self – The updated object.

Return type

object

to_local()

Create regular version of `lightgbm.LGBMRegressor` from the distributed version.

Returns

model – Local underlying model.

Return type

`lightgbm.LGBMRegressor`

9.4.3 `lightgbm.DaskLGBMRanker`

```
class lightgbm.DaskLGBMRanker(*, 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=None, importance_type='split', client=None,
                               **kwargs)
```

Bases: `LGBMRanker`, `_DaskLGBMModel`

Distributed version of `lightgbm.LGBMRanker`.

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

Construct a gradient boosting model.

Parameters

- **boosting_type** (str, optional (default='gbdt')) – ‘gbdt’, traditional Gradient Boosting Decision Tree. ‘dart’, Dropouts meet Multiple Additive Regression Trees. ‘rf’, Random Forest.
- **num_leaves** (int, optional (default=31)) – Maximum tree leaves for base learners.
- **max_depth** (int, optional (default=-1)) – Maximum tree depth for base learners, <=0 means no limit. If setting this to a positive value, consider also changing `num_leaves` to <= 2^{\max_depth} .
- **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** (*str, callable or None, optional (default=None)*) – Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below). Default: ‘regression’ for LGBMRegressor, ‘binary’ or ‘multiclass’ for LGBMClassifier, ‘lambdarank’ for LGBMRanker.
- **class_weight** (*dict, ‘balanced’ or None, optional (default=None)*) – Weights associated with classes in the form {class_label: weight}. Use this parameter only for multi-class classification task; for binary classification task you may use `is_unbalance` or `scale_pos_weight` parameters. Note, that the usage of all these parameters will result in poor estimates of the individual class probabilities. You may want to consider performing probability calibration (<https://scikit-learn.org/stable/modules/calibration.html>) of your model. The ‘balanced’ mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`. If `None`, all classes are supposed to have weight one. Note, that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.
- **min_split_gain** (*float, optional (default=0.)*) – Minimum loss reduction required to make a further partition on a leaf node of the tree.
- **min_child_weight** (*float, optional (default=1e-3)*) – Minimum sum of instance weight (Hessian) needed in a child (leaf).
- **min_child_samples** (*int, optional (default=20)*) – Minimum number of data needed in a child (leaf).
- **subsample** (*float, optional (default=1.)*) – Subsample ratio of the training instance.
- **subsample_freq** (*int, optional (default=0)*) – Frequency of subsample, ≤ 0 means no enable.
- **colsample_bytree** (*float, optional (default=1.)*) – Subsample ratio of columns when constructing each tree.
- **reg_alpha** (*float, optional (default=0.)*) – L1 regularization term on weights.
- **reg_lambda** (*float, optional (default=0.)*) – L2 regularization term on weights.
- **random_state** (*int, RandomState object or None, optional (default=None)*) – Random number seed. If `int`, this number is used to seed the C++ code. If `RandomState` or `Generator` object (`numpy`), a random integer is picked based on its state to seed the C++ code. If `None`, default seeds in C++ code are used.
- **n_jobs** (*int or None, optional (default=None)*) – Number of parallel threads to use for training (can be changed at prediction time by passing it as an extra keyword argument).

For better performance, it is recommended to set this to the number of physical cores in the CPU.

Negative integers are interpreted as following joblib's formula ($n_cpus + 1 + n_jobs$), just like scikit-learn (so e.g. -1 means using all threads). A value of zero corresponds the default number of threads configured for OpenMP in the system. A value of `None` (the default) corresponds to using the number of physical cores in the system (its correct detection requires either the `joblib` or the `psutil` util libraries to be installed).

Changed in version 4.0.0.

- `importance_type` (`str, optional (default='split')`) – The type of feature importance to be filled into `feature_importances_`. If ‘split’, result contains numbers of times the feature is used in a model. If ‘gain’, result contains total gains of splits which use the feature.
- `client` (`dask.distributed.Client or None, optional (default=None)`) – Dask client. If `None`, `distributed.default_client()` will be used at runtime. The Dask client used by this class will not be saved if the model object is pickled.
- `**kwargs` – Other parameters for the model. Check <http://lightgbm.readthedocs.io/en/latest/Parameters.html> for more parameters.

 **Warning**

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

 **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`, `objective(y_true, y_pred, weight) -> grad, hess` or `objective(y_true, y_pred, weight, group) -> grad, hess`:

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. Predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task.

weight

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

group

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

grad

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The value of the first order derivative (gradient) of the loss with respect to the elements of `y_pred` for each sample point.

hess

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The value of the second order derivative (Hessian) of the loss with respect to the elements of y_pred for each sample point.

For multi-class task, y_pred is a numpy 2-D array of shape = [n_samples, n_classes], and grad and hess should be returned in the same format.

Methods

<code>__init__(*[..., boosting_type, num_leaves, ...])</code>	Construct a gradient boosting model.
<code>fit(X, y[, sample_weight, init_score, ...])</code>	Build a gradient boosting model from the training set (X, y).
<code>get_metadata_routing()</code>	Get metadata routing of this object.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, raw_score, start_iteration, ...])</code>	Return the predicted value for each sample.
<code>set_fit_request(*[, eval_at, eval_group, ...])</code>	Configure whether metadata should be requested to be passed to the <code>fit</code> method.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>set_predict_request(*[, num_iteration, ...])</code>	Configure whether metadata should be requested to be passed to the <code>predict</code> method.
<code>to_local()</code>	Create regular version of <code>lightgbm.LGBMRanker</code> from the distributed version.

Attributes

<code>best_iteration_</code>	The best iteration of fitted model if <code>early_stopping()</code> callback has been specified.
<code>best_score_</code>	The best score of fitted model.
<code>booster_</code>	The underlying Booster of this model.
<code>client_</code>	Dask client.
<code>evals_result_</code>	The evaluation results if validation sets have been specified.
<code>feature_importances_</code>	The feature importances (the higher, the more important).
<code>feature_name_</code>	The names of features.
<code>feature_names_in_</code>	scikit-learn compatible version of <code>feature_name_</code> .
<code>n_estimators_</code>	True number of boosting iterations performed.
<code>n_features_</code>	The number of features of fitted model.
<code>n_features_in_</code>	The number of features of fitted model.
<code>n_iter_</code>	True number of boosting iterations performed.
<code>objective_</code>	The concrete objective used while fitting this model.

property best_iteration_

The best iteration of fitted model if `early_stopping()` callback has been specified.

Type

int

property best_score_

The best score of fitted model.

Type
dict

property booster_

The underlying Booster of this model.

Type
Booster

property client_

Dask client.

This property can be passed in the constructor or updated with `model.set_params(client=client)`.

Type
dask.distributed.Client

property evals_result_

The evaluation results if validation sets have been specified.

Type
dict

property feature_importances_

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

Note

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

Type

array of shape = [n_features]

property feature_name_

The names of features.

Note

If input does not contain feature names, they will be added during fitting in the format `Column_0`, `Column_1`, ..., `Column_N`.

Type

list of shape = [n_features]

property feature_names_in_

scikit-learn compatible version of `.feature_name_`.

Added in version 4.5.0.

Type

array of shape = [n_features]

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, 2, 3, 4, 5)`, `**kwargs`)

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

Parameters

- **X** *(Dask Array or Dask DataFrame of shape = [n_samples, n_features])* – Input feature matrix.
- **y** *(Dask Array, Dask DataFrame or Dask Series of shape = [n_samples])* – The target values (class labels in classification, real numbers in regression).
- **sample_weight** *(Dask Array or Dask Series of shape = [n_samples] or None, optional (default=None))* – Weights of training data. Weights should be non-negative.
- **init_score** *(Dask Array or Dask Series of shape = [n_samples] or None, optional (default=None))* – Init score of training data.
- **group** *(Dask Array or Dask Series or None, optional (default=None))* – Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.
- **eval_set** *(list or None, optional (default=None))* – A list of (X, y) tuple pairs to use as validation sets.
- **eval_names** *(list of str, or None, optional (default=None))* – Names of eval_set.
- **eval_sample_weight** *(list of Dask Array or Dask Series, or None, optional (default=None))* – Weights of eval data. Weights should be non-negative.
- **eval_init_score** *(list of Dask Array or Dask Series, or None, optional (default=None))* – Init score of eval data.
- **eval_group** *(list of Dask Array or Dask Series, or None, optional (default=None))* – Group data of eval data.
- **eval_metric** *(str, callable, list or None, optional (default=None))* – If str, it should be a built-in evaluation metric to use. If callable, it should be a custom evaluation metric, see note below for more details. If list, it can be a list of built-in metrics, a list of custom evaluation metrics, or a mix of both. In either case, the metric from the model parameters will be evaluated and used as well. Default: ‘l2’ for LGBMRegressor, ‘logloss’ for LGBMClassifier, ‘ndcg’ for LGBMRanker.
- **eval_at** *(list or tuple of int, optional (default=(1, 2, 3, 4, 5)))* – The evaluation positions of the specified metric.
- **feature_name** *(list of str, or ‘auto’, optional (default='auto'))* – Feature names. If ‘auto’ and data is pandas DataFrame, data columns names are used.
- **categorical_feature** *(list of str or int, or ‘auto’, optional (default='auto'))* – Categorical features. If list of int, interpreted as indices. If list of str, interpreted as feature names (need to specify `feature_name` as well). If ‘auto’ and data is pandas DataFrame, pandas unordered categorical columns are used. All values in categorical features will be cast to int32 and thus should be less than int32

max value (2147483647). Large values could be memory consuming. Consider using consecutive integers starting from zero. All negative values in categorical features will be treated as missing values. The output cannot be monotonically constrained with respect to a categorical feature. Floating point numbers in categorical features will be rounded towards 0.

- ****kwargs** – Other parameters passed through to `LGBMRanker.fit()`.

Returns

self – Returns self.

Return type

`lightgbm.DaskLGBMRanker`

Note

Custom eval function expects a callable with following signatures: `func(y_true, y_pred)`, `func(y_true, y_pred, weight)` or `func(y_true, y_pred, weight, group)` and returns `(eval_name, eval_result, is_higher_better)` or list of `(eval_name, eval_result, is_higher_better)`:

y_true

[numpy 1-D array of shape = [n_samples]] The target values.

y_pred

[numpy 1-D array of shape = [n_samples] or numpy 2-D array of shape = [n_samples, n_classes] (for multi-class task)] The predicted values. In case of custom `objective`, predicted values are returned before any transformation, e.g. they are raw margin instead of probability of positive class for binary task in this case.

weight

[numpy 1-D array of shape = [n_samples]] The weight of samples. Weights should be non-negative.

group

[numpy 1-D array] Group/query data. Only used in the learning-to-rank task. `sum(group) = n_samples`. For example, if you have a 100-document dataset with `group = [10, 20, 40, 10, 10, 10]`, that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, records 31-70 are in the third group, etc.

eval_name

[str] The name of evaluation function (without whitespace).

eval_result

[float] The eval result.

is_higher_better

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

`get_metadata_routing()`

Get metadata routing of this object.

Please check [User Guide](#) on how the routing mechanism works.

Returns

routing – A `MetadataRequest` encapsulating routing information.

Return type

`MetadataRequest`

get_params(deep=True)

Get parameters for this estimator.

Parameters

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

Returns

params – Parameter names mapped to their values.

Return type

dict

property n_estimators_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property n_features_

The number of features of fitted model.

Type

int

property n_features_in_

The number of features of fitted model.

Type

int

property n_iter_

True number of boosting iterations performed.

This might be less than parameter `n_estimators` if early stopping was enabled or if boosting stopped early due to limits on complexity like `min_gain_to_split`.

Added in version 4.0.0.

Type

int

property objective_

The concrete objective used while fitting this model.

Type

str or callable

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

Return the predicted value for each sample.

Parameters

- **X** (Dask Array or Dask DataFrame of shape = [n_samples, n_features]) – Input features matrix.

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

 **Note**

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

- **validate_features** (bool, optional (default=False)) – If True, ensure that the features used to predict match the ones used to train. Used only if data is pandas DataFrame.
- ****kwargs** – Other parameters for the prediction.

Returns

- **predicted_result** (Dask Array of shape = [n_samples]) – The predicted values.
- **X_leaves** (Dask Array of shape = [n_samples, n_trees]) – If **pred_leaf**=True, the predicted leaf of every tree for each sample.
- **X_SHAP_values** (Dask Array of shape = [n_samples, n_features + 1]) – If **pred_contrib**=True, the feature contributions for each sample.

```
set_fit_request(*, eval_at='UNCHANGED$', eval_group='UNCHANGED$',
                eval_init_score='UNCHANGED$', eval_metric='UNCHANGED$',
                eval_names='UNCHANGED$', eval_sample_weight='UNCHANGED$',
                eval_set='UNCHANGED$', group='UNCHANGED$', init_score='UNCHANGED$',
                sample_weight='UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the **fit** method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- True: metadata is requested, and passed to **fit** if provided. The request is ignored if metadata is not provided.
- False: metadata is not requested and the meta-estimator will not pass it to **fit**.
- None: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- str: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- **eval_at** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_at` parameter in `fit`.
- **eval_group** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_group` parameter in `fit`.
- **eval_init_score** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_init_score` parameter in `fit`.
- **eval_metric** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_metric` parameter in `fit`.
- **eval_names** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_names` parameter in `fit`.
- **eval_sample_weight** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_sample_weight` parameter in `fit`.
- **eval_set** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `eval_set` parameter in `fit`.
- **group** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `group` parameter in `fit`.
- **init_score** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `init_score` parameter in `fit`.
- **sample_weight** (str, True, False, or None, default=`sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `sample_weight` parameter in `fit`.

Returns

`self` – The updated object.

Return type

object

`set_params(**params)`

Set the parameters of this estimator.

Parameters

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

Returns

`self` – Returns self.

Return type

object

```
set_predict_request(*, num_iteration='UNCHANGED$', pred_contrib='UNCHANGED$',  
pred_leaf='UNCHANGED$', raw_score='UNCHANGED$',  
start_iteration='UNCHANGED$', validate_features='UNCHANGED$')
```

Configure whether metadata should be requested to be passed to the `predict` method.

Note that this method is only relevant when this estimator is used as a sub-estimator within a `meta-estimator` and metadata routing is enabled with `enable_metadata_routing=True` (see `sklearn.set_config()`). Please check the [User Guide](#) on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `predict` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `predict`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Parameters

- `num_iteration` (`str`, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `num_iteration` parameter in `predict`.
- `pred_contrib` (`str`, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_contrib` parameter in `predict`.
- `pred_leaf` (`str`, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `pred_leaf` parameter in `predict`.
- `raw_score` (`str`, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `raw_score` parameter in `predict`.
- `start_iteration` (`str`, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `start_iteration` parameter in `predict`.
- `validate_features` (`str`, `True`, `False`, or `None`, `default=sklearn.utils.metadata_routing.UNCHANGED`) – Metadata routing for `validate_features` parameter in `predict`.

Returns

`self` – The updated object.

Return type

object

to_local()

Create regular version of `lightgbm.LGBMRanker` from the distributed version.

Returns

`model` – Local underlying model.

Return type

`lightgbm.LGBMRanker`

9.5 Callbacks

<code>early_stopping(stopping_rounds[, ...])</code>	Create a callback that activates early stopping.
<code>log_evaluation([period, show_stdv])</code>	Create a callback that logs the evaluation results.
<code>record_evaluation(eval_result)</code>	Create a callback that records the evaluation history into <code>eval_result</code> .
<code>reset_parameter(**kwargs)</code>	Create a callback that resets the parameter after the first iteration.

9.5.1 `lightgbm.early_stopping`

`lightgbm.early_stopping(stopping_rounds, first_metric_only=False, verbose=True, min_delta=0.0)`

Create a callback that activates early stopping.

Activates early stopping. The model will train until the validation score doesn't improve by at least `min_delta`. Validation score needs to improve at least every `stopping_rounds` round(s) to continue training. Requires at least one validation data and one metric. If there's more than one, will check all of them. But the training data is ignored anyway. To check only the first metric set `first_metric_only` to True. The index of iteration that has the best performance will be saved in the `best_iteration` attribute of a model.

Note

If using `boosting_type="dart"`, this callback has no effect and early stopping will not be performed.

Parameters

- `stopping_rounds (int)` – The possible number of rounds without the trend occurrence.
- `first_metric_only (bool, optional (default=False))` – Whether to use only the first metric for early stopping.
- `verbose (bool, optional (default=True))` – Whether to log message with early stopping information. By default, standard output resource is used. Use `register_logger()` function to register a custom logger.
- `min_delta (float or list of float, optional (default=0.0))` – Minimum improvement in score to keep training. If float, this single value is used for all metrics. If list, its length should match the total number of metrics.

Added in version 4.0.0.

Returns

`callback` – The callback that activates early stopping.

Return type

`_EarlyStoppingCallback`

9.5.2 `lightgbm.log_evaluation`

`lightgbm.log_evaluation(period=1, show_stdv=True)`

Create a callback that logs the evaluation results.

By default, standard output resource is used. Use `register_logger()` function to register a custom logger.

 **Note**

Requires at least one validation data.

Parameters

- `period(int, optional (default=1))` – The period to log the evaluation results. The last boosting stage or the boosting stage found by using `early_stopping` callback is also logged.
- `show_stdv(bool, optional (default=True))` – Whether to log stdv (if provided).

Returns

`callback` – The callback that logs the evaluation results every `period` boosting iteration(s).

Return type

`_LogEvaluationCallback`

9.5.3 `lightgbm.record_evaluation`

`lightgbm.record_evaluation(eval_result)`

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

Parameters

`eval_result (dict)` – Dictionary used to store all evaluation results of all validation sets. This should be initialized outside of your call to `record_evaluation()` and should be empty. Any initial contents of the dictionary will be deleted.

Example

With two validation sets named ‘eval’ and ‘train’, and one evaluation metric named ‘logloss’ this dictionary after finishing a model training process will have the following structure:

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

Returns

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

Return type

`_RecordEvaluationCallback`

9.5.4 `lightgbm.reset_parameter`

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

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

Note

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

Parameters

`**kwargs` (*value should be list or callable*) – List of parameters for each boosting round or a callable 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 callable func, parameter = func(current_round).

Returns

`callback` – The callback that resets the parameter after the first iteration.

Return type

`_ResetParameterCallback`

9.6 Plotting

<code>plot_importance</code> (booster[, ax, height, xlim, ...])	Plot model's feature importances.
<code>plot_split_value_histogram</code> (booster, feature)	Plot split value histogram for the specified feature of the model.
<code>plot_metric</code> (booster[, metric, ...])	Plot one metric during training.
<code>plot_tree</code> (booster[, ax, tree_index, ...])	Plot specified tree.
<code>create_tree_digraph</code> (booster[, tree_index, ...])	Create a digraph representation of specified tree.

9.6.1 `lightgbm.plot_importance`

```
lightgbm.plot_importance(booster, ax=None, height=0.2, xlim=None, ylim=None, title='Feature importance',  
                        xlabel='Feature importance', ylabel='Features', importance_type='auto',  
                        max_num_features=None, ignore_zero=True, figsize=None, dpi=None, grid=True,  
                        precision=3, **kwargs)
```

Plot model's feature importances.

Parameters

- **booster** (`Booster` or `LGBMModel`) – Booster or LGBMModel instance which feature importance should be plotted.
- **ax** (`matplotlib.axes.Axes` or `None`, optional (`default=None`)) – Target axes instance. If `None`, new figure and axes will be created.
- **height** (`float`, optional (`default=0.2`)) – Bar height, passed to `ax.bars()`.
- **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** (str or None, optional (default="Feature importance")) – Axes title. If None, title is disabled.
- **xlabel** (str or None, optional (default="Feature importance")) – X-axis title label. If None, title is disabled. @importance_type@ placeholder can be used, and it will be replaced with the value of `importance_type` parameter.
- **ylabel** (str or None, optional (default="Features")) – Y-axis title label. If None, title is disabled.
- **importance_type** (str, optional (default="auto")) – How the importance is calculated. If "auto", if booster parameter is `LGBMModel`, `booster.importance_type` attribute is used; "split" otherwise. If "split", result contains numbers of times the feature is used in a model. If "gain", result contains total gains of splits which use the feature.
- **max_num_features** (int or None, optional (default=None)) – Max number of top features displayed on plot. If None or <1, all features will be displayed.
- **ignore_zero** (bool, optional (default=True)) – Whether to ignore features with zero importance.
- **figsize** (tuple of 2 elements or None, optional (default=None)) – Figure size.
- **dpi** (int or None, optional (default=None)) – Resolution of the figure.
- **grid** (bool, optional (default=True)) – Whether to add a grid for axes.
- **precision** (int or None, optional (default=3)) – Used to restrict the display of floating point values to a certain precision.
- ****kwargs** – Other parameters passed to `ax.barh()`.

Returns

`ax` – The plot with model's feature importances.

Return type

`matplotlib.axes.Axes`

9.6.2 `lightgbm.plot_split_value_histogram`

```
lightgbm.plot_split_value_histogram(booster, feature, bins=None, ax=None, width_coef=0.8, xlim=None,
                                    ylim=None, title='Split value histogram for feature with
                                    @index/name@ @feature@', xlabel='Feature split value',
                                    ylabel='Count', figsize=None, dpi=None, grid=True, **kwargs)
```

Plot split value histogram for the specified feature of the model.

Parameters

- **booster** (Booster or `LGBMModel`) – Booster or `LGBMModel` instance of which feature split value histogram should be plotted.
- **feature** (int or str) – The feature name or index the histogram is plotted for. If int, interpreted as index. If str, interpreted as name.
- **bins** (int, str or None, optional (default=None)) – The maximum number of bins. If None, the number of bins equals number of unique split values. If str, it should be one from the list of the supported values by `numpy.histogram()` function.
- **ax** (matplotlib.axes.Axes or None, optional (default=None)) – Target axes instance. If None, new figure and axes will be created.

- **width_coef** (*float, optional (default=0.8)*) – Coefficient for histogram bar width.
- **xlim** (*tuple of 2 elements or None, optional (default=None)*) – Tuple passed to `ax.xlim()`.
- **ylim** (*tuple of 2 elements or None, optional (default=None)*) – Tuple passed to `ax.ylim()`.
- **title** (*str or None, optional (default="Split value histogram for feature with @index/name@ @feature@")*) – Axes title. If None, title is disabled. `@feature@` placeholder can be used, and it will be replaced with the value of `feature` parameter. `@index/name@` placeholder can be used, and it will be replaced with `index` word in case of `int` type `feature` parameter or `name` word in case of `str` type `feature` parameter.
- **xlabel** (*str or None, optional (default="Feature split value")*) – X-axis title label. If None, title is disabled.
- **ylabel** (*str or None, optional (default="Count")*) – Y-axis title label. If None, title is disabled.
- **figsize** (*tuple of 2 elements or None, optional (default=None)*) – Figure size.
- **dpi** (*int or None, optional (default=None)*) – Resolution of the figure.
- **grid** (*bool, optional (default=True)*) – Whether to add a grid for axes.
- ****kwargs** – Other parameters passed to `ax.bar()`.

Returns

`ax` – The plot with specified model's feature split value histogram.

Return type

`matplotlib.axes.Axes`

9.6.3 `lightgbm.plot_metric`

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

Plot one metric during training.

Parameters

- **booster** (*dict or `LGBMModel`*) – Dictionary returned from `lightgbm.train()` or `LGBMModel` instance.
- **metric** (*str 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 str, 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** (*str or None, optional (default="Metric during training")*) – Axes title. If None, title is disabled.
- **xlabel** (*str or None, optional (default="Iterations")*) – X-axis title label. If None, title is disabled.
- **ylabel** (*str or None, optional (default="@metric@")*) – Y-axis title label. If ‘auto’, metric name is used. If None, title is disabled. @metric@ placeholder can be used, and it will be replaced with metric name.
- **figsize** (*tuple of 2 elements or None, optional (default=None)*) – Figure size.
- **dpi** (*int or None, optional (default=None)*) – Resolution of the figure.
- **grid** (*bool, optional (default=True)*) – Whether to add a grid for axes.

Returns

`ax` – The plot with metric’s history over the training.

Return type

`matplotlib.axes.Axes`

9.6.4 `lightgbm.plot_tree`

```
lightgbm.plot_tree(booster, ax=None, tree_index=0, figsize=None, dpi=None, show_info=None, precision=3, orientation='horizontal', example_case=None, **kwargs)
```

Plot specified tree.

Each node in the graph represents a node in the tree.

Non-leaf nodes have labels like `Column_10 <= 875.9`, which means “this node splits on the feature named “`Column_10`”, with threshold `875.9`”.

Leaf nodes have labels like `leaf 2: 0.422`, which means “this node is a leaf node, and the predicted value for records that fall into this node is `0.422`”. The number (2) is an internal unique identifier and doesn’t have any special meaning.

 **Note**

It is preferable to use `create_tree_digraph()` because of its lossless quality and returned objects can be also rendered and displayed directly inside a Jupyter notebook.

Parameters

- **booster** (`Booster` or `LGBMModel`) – Booster or LGBMModel instance to be plotted.
- **ax** (`matplotlib.axes.Axes` or `None`, optional (default=None)) – Target axes instance. If None, new figure and axes will be created.
- **tree_index** (*int, optional (default=0)*) – The index of a target tree to plot.
- **figsize** (*tuple of 2 elements or None, optional (default=None)*) – Figure size.
- **dpi** (*int or None, optional (default=None)*) – Resolution of the figure.

- **show_info** (*list of str, or None, optional (default=None)*) – What information should be shown in nodes.
 - 'split_gain' : gain from adding this split to the model
 - 'internal_value' : raw predicted value that would be produced by this node if it was a leaf node
 - 'internal_count' : number of records from the training data that fall into this non-leaf node
 - 'internal_weight' : total weight of all nodes that fall into this non-leaf node
 - 'leaf_count' : number of records from the training data that fall into this leaf node
 - 'leaf_weight' : total weight (sum of Hessian) of all observations that fall into this leaf node
 - 'data_percentage' : percentage of training data that fall into this node
- **precision** (*int or None, optional (default=3)*) – Used to restrict the display of floating point values to a certain precision.
- **orientation** (*str, optional (default='horizontal')*) – Orientation of the tree. Can be ‘horizontal’ or ‘vertical’.
- **example_case** (*numpy 2-D array, pandas DataFrame or None, optional (default=None)*) – Single row with the same structure as the training data. If not None, the plot will highlight the path that sample takes through the tree.

Added in version 4.0.0.

- ****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`

9.6.5 `lightgbm.create_tree_digraph`

```
lightgbm.create_tree_digraph(booster, tree_index=0, show_info=None, precision=3,
                           orientation='horizontal', example_case=None, max_category_values=10,
                           **kwargs)
```

Create a digraph representation of specified tree.

Each node in the graph represents a node in the tree.

Non-leaf nodes have labels like `Column_10 <= 875.9`, which means “this node splits on the feature named “`Column_10`”, with threshold `875.9`”.

Leaf nodes have labels like `leaf 2: 0.422`, which means “this node is a leaf node, and the predicted value for records that fall into this node is `0.422`”. The number (2) is an internal unique identifier and doesn’t have any special meaning.

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 str`, or `None`, optional (default=None)) – What information should be shown in nodes.
 - 'split_gain' : gain from adding this split to the model
 - 'internal_value' : raw predicted value that would be produced by this node if it was a leaf node
 - 'internal_count' : number of records from the training data that fall into this non-leaf node
 - 'internal_weight' : total weight of all nodes that fall into this non-leaf node
 - 'leaf_count' : number of records from the training data that fall into this leaf node
 - 'leaf_weight' : total weight (sum of Hessian) of all observations that fall into this leaf node
 - 'data_percentage' : percentage of training data that fall into this node
- **precision** (`int` or `None`, optional (default=3)) – Used to restrict the display of floating point values to a certain precision.
- **orientation** (`str`, optional (default='horizontal')) – Orientation of the tree. Can be 'horizontal' or 'vertical'.
- **example_case** (`numpy 2-D array`, `pandas DataFrame` or `None`, optional (default=None)) – Single row with the same structure as the training data. If not None, the plot will highlight the path that sample takes through the tree.

Added in version 4.0.0.

- **max_category_values** (`int`, optional (default=10)) – The maximum number of category values to display in tree nodes, if the number of thresholds is greater than this value, thresholds will be collapsed and displayed on the label tooltip instead.

⚠ Warning

Consider wrapping the SVG string of the tree graph with `IPython.display.HTML` when running on JupyterLab to get the `tooltip` working right.

Example:

```
from IPython.display import HTML

graph = lgb.create_tree_digraph(clf, max_category_values=5)
HTML(graph._repr_image_svg_xml())
```

Added in version 4.0.0.

- ****kwargs** – Other parameters passed to `Digraph` constructor. Check <https://graphviz.readthedocs.io/en/stable/api.html#digraph> for the full list of supported parameters.

Returns

`graph` – The digraph representation of specified tree.

Return type
graphviz.Digraph

9.7 Utilities

<code>register_logger(logger[, info_method_name, ...])</code>	Register custom logger.
---	-------------------------

9.7.1 `lightgbm.register_logger`

`lightgbm.register_logger(logger, info_method_name='info', warning_method_name='warning')`

Register custom logger.

Parameters

- **logger** (*Any*) – Custom logger.
- **info_method_name** (*str, optional (default="info")*) – Method used to log info messages.
- **warning_method_name** (*str, optional (default="warning")*) – Method used to log warning messages.

DISTRIBUTED LEARNING GUIDE

This guide describes distributed learning in LightGBM. Distributed learning allows the use of multiple machines to produce a single model.

Follow the [Quick Start](#) to know how to use LightGBM first.

10.1 How Distributed LightGBM Works

This section describes how distributed learning in LightGBM works. To learn how to do this in various programming languages and frameworks, please see [Integrations](#).

10.1.1 Choose Appropriate Parallel Algorithm

LightGBM provides 3 distributed learning algorithms now.

Parallel Algorithm	How to Use
Data parallel	<code>tree_learner=data</code>
Feature parallel	<code>tree_learner=feature</code>
Voting parallel	<code>tree_learner=voting</code>

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 distributed learning](#).

10.2 Integrations

This section describes how to run distributed LightGBM training in various programming languages and frameworks. To learn how distributed learning in LightGBM works generally, please see [How Distributed LightGBM Works](#).

10.2.1 Apache Spark

Apache Spark users can use [SynapseML](#) for machine learning workflows with LightGBM. This project is not maintained by LightGBM's maintainers.

See this [SynapseML example](#) for additional information on using LightGBM on Spark.

 **Note**

SynapseML is not maintained by LightGBM's maintainers. Bug reports or feature requests should be directed to <https://github.com/microsoft/SynapseML/issues>.

10.2.2 Dask

Added in version 3.2.0.

LightGBM's Python-package supports distributed learning via [Dask](#). This integration is maintained by LightGBM's maintainers.

 **Warning**

Dask integration is only tested on macOS and Linux.

Dask Examples

For sample code using `lightgbm.dask`, see these [Dask examples](#).

Training with Dask

This section contains detailed information on performing LightGBM distributed training using Dask.

Configuring the Dask Cluster

Allocating Threads

When setting up a Dask cluster for training, give each Dask worker process at least two threads. If you do not do this, training might be substantially slower because communication work and training work will block each other.

If you do not have other significant processes competing with Dask for resources, just accept the default `nthreads` from your chosen `dask.distributed` cluster.

```
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=3)
client = Client(cluster)
```

Managing Memory

Use the Dask diagnostic dashboard or your preferred monitoring tool to monitor Dask workers' memory consumption during training. As described in [the Dask worker documentation](#), Dask workers will automatically start spilling data to disk if memory consumption gets too high. This can substantially slow down computations, since disk I/O is usually much slower than reading the same data from memory.

At 60% of memory load, [Dask will] spill least recently used data to disk

To reduce the risk of hitting memory limits, consider restarting each worker process before running any data loading or training code.

```
client.restart()
```

Setting Up Training Data

The estimators in `lightgbm.dask` expect that matrix-like or array-like data are provided in Dask DataFrame, Dask Array, or (in some cases) Dask Series format. See the [Dask DataFrame documentation](#) and the [Dask Array documentation](#) for more information on how to create such data structures.

While setting up for training, `lightgbm` will concatenate all of the partitions on a worker into a single dataset. Distributed training then proceeds with one LightGBM worker process per Dask worker.

When setting up data partitioning for LightGBM training with Dask, try to follow these suggestions:

- ensure that each worker in the cluster has some of the training data
- try to give each worker roughly the same amount of data, especially if your dataset is small
- if you plan to train multiple models (for example, to tune hyperparameters) on the same data, use `client.persist()` before training to materialize the data one time

Using a Specific Dask Client

In most situations, you should not need to tell `lightgbm.dask` to use a specific Dask client. By default, the client returned by `distributed.default_client()` will be used.

However, you might want to explicitly control the Dask client used by LightGBM if you have multiple active clients in the same session. This is useful in more complex workflows like running multiple training jobs on different Dask clusters.

LightGBM's Dask estimators support setting an attribute `client` to control the client that is used.

```
import lightgbm as lgb
from distributed import Client, LocalCluster

cluster = LocalCluster()
client = Client(cluster)

# option 1: keyword argument in constructor
dask_model = lgb.DaskLGBMClassifier(client=client)

# option 2: set_params() after construction
dask_model = lgb.DaskLGBMClassifier()
dask_model.set_params(client=client)
```

Using Specific Ports

At the beginning of training, `lightgbm.dask` sets up a LightGBM network where each Dask worker runs one long-running task that acts as a LightGBM worker. During training, LightGBM workers communicate with each other over TCP sockets. By default, random open ports are used when creating these sockets.

If the communication between Dask workers in the cluster used for training is restricted by firewall rules, you must tell LightGBM exactly what ports to use.

Option 1: provide a specific list of addresses and ports

LightGBM supports a parameter `machines`, a comma-delimited string where each entry refers to one worker (host name or IP) and a port that that worker will accept connections on. If you provide this parameter to the estimators in `lightgbm.dask`, LightGBM will not search randomly for ports.

For example, consider the case where you are running one Dask worker process on each of the following IP addresses:

```
10.0.1.0
10.0.2.0
10.0.3.0
```

You could edit your firewall rules to allow traffic on one additional port on each of these hosts, then provide `machines` directly.

```
import lightgbm as lgb

machines = "10.0.1.0:12401,10.0.2.0:12402,10.0.3.0:15000"
dask_model = lgb.DaskLGBMRegressor(machines=machines)
```

If you are running multiple Dask worker processes on physical host in the cluster, be sure that there are multiple entries for that IP address, with different ports. For example, if you were running a cluster with `nprocs=2` (2 Dask worker processes per machine), you might open two additional ports on each of these hosts, then provide `machines` as follows.

```
import lightgbm as lgb

machines = ",".join([
    "10.0.1.0:16000",
    "10.0.1.0:16001",
    "10.0.2.0:16000",
    "10.0.2.0:16001",
])
dask_model = lgb.DaskLGBMRegressor(machines=machines)
```

Warning

Providing `machines` gives you complete control over the networking details of training, but it also makes the training process fragile. Training will fail if you use `machines` and any of the following are true:

- any of the ports mentioned in `machines` are not open when training begins
- some partitions of the training data are held by machines that are not present in `machines`
- some machines mentioned in `machines` do not hold any of the training data

Option 2: specify one port to use on every worker

If you are only running one Dask worker process on each host, and if you can reliably identify a port that is open on every host, using `machines` is unnecessarily complicated. If `local_listen_port` is given and `machines` is not, LightGBM will not search for ports randomly, but it will limit the list of addresses in the LightGBM network to those Dask workers that have a piece of the training data.

For example, consider the case where you are running one Dask worker process on each of the following IP addresses:

```
10.0.1.0
10.0.2.0
10.0.3.0
```

You could edit your firewall rules to allow communication between any of the workers over one port, then provide that port via parameter `local_listen_port`.

```
import lightgbm as lgb

dask_model = lgb.DaskLGBMRegressor(local_listen_port=12400)
```

 **Warning**

Providing `local_listen_port` is slightly less fragile than `machines` because LightGBM will automatically figure out which workers have pieces of the training data. However, using this method, training can fail if any of the following are true:

- the port `local_listen_port` is not open on any of the worker hosts
- any machine has multiple Dask worker processes running on it

Using Custom Objective Functions with Dask

Added in version 4.0.0.

It is possible to customize the boosting process by providing a custom objective function written in Python. See the Dask API's documentation for details on how to implement such functions.

 **Warning**

Custom objective functions used with `lightgbm.dask` will be called by each worker process on only that worker's local data.

Follow the example below to use a custom implementation of the `regression_l2` objective.

```
import dask.array as da
import lightgbm as lgb
import numpy as np
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=2)
client = Client(cluster)

X = da.random.random((1000, 10), (500, 10))
y = da.random.random((1000,), (500,))

def custom_l2_obj(y_true, y_pred):
    grad = y_pred - y_true
    hess = np.ones(len(y_true))
    return grad, hess

dask_model = lgb.DaskLGBMRegressor(
    objective=custom_l2_obj
)
dask_model.fit(X, y)
```

Prediction with Dask

The estimators from `lightgbm.dask` can be used to create predictions based on data stored in Dask collections. In that interface, `.predict()` expects a Dask Array or Dask DataFrame, and returns a Dask Array of predictions.

See the [Dask prediction example](#) for some sample code that shows how to perform Dask-based prediction.

For model evaluation, consider using [the metrics functions from dask-ml](#). Those functions are intended to provide the same API as equivalent functions in `sklearn.metrics`, but they use distributed computation powered by Dask to compute metrics without all of the input data ever needing to be on a single machine.

Saving Dask Models

After training with Dask, you have several options for saving a fitted model.

Option 1: pickle the Dask estimator

LightGBM's Dask estimators can be pickled directly with `cloudpickle`, `joblib`, or `pickle`.

```
import dask.array as da
import pickle
import lightgbm as lgb
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=2)
client = Client(cluster)

X = da.random.random((1000, 10), (500, 10))
y = da.random.random((1000,), (500,))

dask_model = lgb.DaskLGBMRegressor()
dask_model.fit(X, y)

with open("dask-model.pkl", "wb") as f:
    pickle.dump(dask_model, f)
```

A model saved this way can then later be loaded with whichever serialization library you used to save it.

```
import pickle
with open("dask-model.pkl", "rb") as f:
    dask_model = pickle.load(f)
```

Note

If you explicitly set a Dask client (see [Using a Specific Dask Client](#)), it will not be saved when pickling the estimator. When loading a Dask estimator from disk, if you need to use a specific client you can add it after loading with `dask_model.set_params(client=client)`.

Option 2: pickle the sklearn estimator

The estimators available from `lightgbm.dask` can be converted to an instance of the equivalent class from `lightgbm.sklearn`. Choosing this option allows you to use Dask for training but avoid depending on any Dask libraries at scoring time.

```
import dask.array as da
import joblib
```

(continues on next page)

(continued from previous page)

```

import lightgbm as lgb
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=2)
client = Client(cluster)

X = da.random.random((1000, 10), (500, 10))
y = da.random.random((1000,), (500,))

dask_model = lgb.DaskLGBMRegressor()
dask_model.fit(X, y)

# convert to sklearn equivalent
sklearn_model = dask_model.to_local()

print(type(sklearn_model))
#> lightgbm.sklearn.LGBMRegressor

joblib.dump(sklearn_model, "sklearn-model.joblib")

```

A model saved this way can then later be loaded with whichever serialization library you used to save it.

```

import joblib

sklearn_model = joblib.load("sklearn-model.joblib")

```

Option 3: save the LightGBM Booster

The lowest-level model object in LightGBM is the `lightgbm.Booster`. After training, you can extract a Booster from the Dask estimator.

```

import dask.array as da
import lightgbm as lgb
from distributed import Client, LocalCluster

cluster = LocalCluster(n_workers=2)
client = Client(cluster)

X = da.random.random((1000, 10), (500, 10))
y = da.random.random((1000,), (500,))

dask_model = lgb.DaskLGBMRegressor()
dask_model.fit(X, y)

# get underlying Booster object
bst = dask_model.booster_

```

From the point forward, you can use any of the following methods to save the Booster:

- serialize with `cloudpickle`, `joblib`, or `pickle`
- `bst.dump_model()`: dump the model to a dictionary which could be written out as JSON
- `bst.model_to_string()`: dump the model to a string in memory
- `bst.save_model()`: write the output of `bst.model_to_string()` to a text file

10.2.3 Kubeflow

Kubeflow users can also use the [Kubeflow XGBoost Operator](#) for machine learning workflows with LightGBM. You can see [this example](#) for more details.

Kubeflow integrations for LightGBM are not maintained by LightGBM's maintainers.

Note

The Kubeflow integrations for LightGBM are not maintained by LightGBM's maintainers. Bug reports or feature requests should be directed to <https://github.com/kubeflow/fairing/issues> or <https://github.com/kubeflow/xgboost-operator/issues>.

10.2.4 LightGBM CLI

Preparation

By default, distributed learning with LightGBM uses socket-based communication.

If you need to build distributed version with MPI support, please refer to [Installation Guide](#).

Socket Version

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

```
machine1_ip 12345
machine2_ip 12345
```

MPI Version

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

```
machine1_ip
machine2_ip
```

Note: For Windows users, need to start “smpd” to start MPI service. More details can be found [here](#).

Run Distributed Learning

Socket Version

1. Edit following parameters in config file:

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

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

`machine_list_file=mlist.txt`, `mlist.txt` is created in [Preparation section](#).

`local_listen_port=12345`, 12345 is allocated in [Preparation section](#).

2. Copy data file, executable file, config file and `mlist.txt` to all machines.

3. Run following command on all machines, you need to change `your_config_file` to real config file.

For Windows: `lightgbm.exe config=your_config_file`

For Linux: `./lightgbm config=your_config_file`

MPI Version

1. Edit following parameters in config file:
`tree_learner=your_parallel_algorithm`, edit `your_parallel_algorithm` (e.g. feature/data) here.
`num_machines=your_num_machines`, edit `your_num_machines` (e.g. 4) here.
2. Copy data file, executable file, config file and `mlist.txt` to all machines.
Note: MPI needs to be run in the **same path on all machines**.
3. Run following command on one machine (not need to run on all machines), need to change `your_config_file` to real config file.

For Windows:

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

For Linux:

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

Example

- A simple distributed learning example

10.2.5 Ray

Ray is a Python-based framework for distributed computing. Ray provides LightGBM support through the Ray Train API with `LightGBMTrainer` and the `lightgbm_ray` project maintained within the official Ray GitHub organization.

For the Ray Train API, see [the Ray documentation](#) for usage examples.

For the `lightgbm_ray` project, see [the lightgbm_ray documentation](#) for usage examples.

Note

`lightgbm_ray` and `ray` are not maintained by LightGBM's maintainers. Bug reports or feature requests should be directed to https://github.com/ray-project/lightgbm_ray/issues and <https://github.com/ray-project/ray/issues> respectively.

10.2.6 Mars

Mars is a tensor-based framework for large-scale data computation. LightGBM integration, maintained within the Mars GitHub repository, can be used to perform distributed LightGBM training using `pymars`.

See [the mars documentation](#) for usage examples.

Note

Mars is not maintained by LightGBM's maintainers. Bug reports or feature requests should be directed to <https://github.com/mars-project/mars/issues>.

LIGHTGBM GPU TUTORIAL

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

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

11.1 GPU Setup

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

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

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

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

After installing the drivers you need to restart the server.

```
sudo init 6
```

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

If you are using an AMD GPU, you should download and install the [AMDGPU-Pro](#) driver and also install packages `ocl-icd-libopencl1` and `ocl-icd-opencl-dev`.

11.2 Build LightGBM

Now install necessary building tools and dependencies:

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

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

```
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
cmake -B build -S . -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 -B build -S . -DUSE_GPU=1 -DOpenCL_LIBRARY=/usr/local/cuda/lib64/libOpenCL.so -
# DOpenCL_INCLUDE_DIR=/usr/local/cuda/include/
cmake --build build -j$(nproc)
```

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

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

11.3 Install Python Interface (optional)

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

```
sudo apt-get -y install python-pip
sudo -H pip install setuptools numpy scipy scikit-learn -U
sudo sh ./build-python.sh install --precompile
```

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

You can read our [Python-package Examples](#) for more information on how to use the Python interface.

11.4 Dataset Preparation

Using the following commands to prepare the Higgs dataset:

```
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
EOF
```

(continues on next page)

(continued from previous page)

```

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
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.

11.5 Run Your First Learning Task on GPU

Now we are ready to start GPU training!

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

```

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

```

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

```

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

```

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

```

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

```

Speed test on CPU:

```

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

```

You should observe over three times speedup on this GPU.

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

```

./lightgbm config=lightgbm_gpu.conf data=higgs.train objective=regression_l2 metric=12

```

Also, you can compare the training speed with CPU:

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

11.6 Further Reading

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

11.7 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.

ADVANCED TOPICS

12.1 Missing Value Handle

- LightGBM enables the missing value handle by default. Disable it by setting `use_missing=false`.
- LightGBM uses NA (NaN) to represent missing values by default. Change it to use zero by setting `zero_as_missing=true`.
- When `zero_as_missing=false` (default), the unrecorded values in sparse matrices (and LightSVM) are treated as zeros.
- When `zero_as_missing=true`, NA and zeros (including unrecorded values in sparse matrices (and LightSVM)) are treated as missing.

12.2 Categorical Feature Support

- LightGBM offers good accuracy with integer-encoded categorical features. LightGBM applies Fisher (1958) to find the optimal split over categories as described [here](#). This often performs better than one-hot encoding.
- Use `categorical_feature` to specify the categorical features. Refer to the parameter `categorical_feature` in [Parameters](#).
- Categorical features will be cast to `int32` (integer codes will be extracted from pandas categoricals in the Python-package) so they must be encoded as non-negative integers (negative values will be treated as missing) less than `Int32.MaxValue` (2147483647). It is best to use a contiguous range of integers started from zero. Floating point numbers in categorical features will be rounded towards 0.
- Use `min_data_per_group`, `cat_smooth` to deal with over-fitting (when `#data` is small or `#category` is large).
- For a categorical feature with high cardinality (`#category` is large), it often works best to treat the feature as numeric, either by simply ignoring the categorical interpretation of the integers or by embedding the categories in a low-dimensional numeric space.

12.3 LambdaRank

- The label should be of type `int`, such that larger numbers correspond to higher relevance (e.g. 0:bad, 1:fair, 2:good, 3:perfect).
- Use `label_gain` to set the gain(weight) of `int` label.
- Use `lambdaRank_truncation_level` to truncate the max DCG.

12.4 Cost Efficient Gradient Boosting

Cost Efficient Gradient Boosting (CEGB) makes it possible to penalise boosting based on the cost of obtaining feature values. CEGB penalises learning in the following ways:

- Each time a tree is split, a penalty of `cegb_penalty_split` is applied.
- When a feature is used for the first time, `cegb_penalty_feature_coupled` is applied. This penalty can be different for each feature and should be specified as one double per feature.
- When a feature is used for the first time for a data row, `cegb_penalty_feature_lazy` is applied. Like `cegb_penalty_feature_coupled`, this penalty is specified as one double per feature.

Each of the penalties above is scaled by `cegb_tradeoff`. Using this parameter, it is possible to change the overall strength of the CEGB penalties by changing only one parameter.

12.5 Parameters Tuning

- Refer to [Parameters Tuning](#).

12.6 Distributed Learning

- Refer to [Distributed Learning Guide](#).

12.7 GPU Support

- Refer to [GPU Tutorial](#) and [GPU Targets](#).

12.8 Support for Position Bias Treatment

Often the relevance labels provided in Learning-to-Rank tasks might be derived from implicit user feedback (e.g., clicks) and therefore might be biased due to their position/location on the screen when having been presented to a user. LightGBM can make use of positional data.

For example, consider the case where you expect that the first 3 results from a search engine will be visible in users' browsers without scrolling, and all other results for a query would require scrolling.

LightGBM could be told to account for the position bias from results being "above the fold" by providing a `positions` array encoded as follows:

```
0
0
0
1
1
0
0
0
1
1
...
...
```

Where 0 = "above the fold" and 1 = "requires scrolling". The specific values are not important, as long as they are consistent across all observations in the training data. An encoding like 100 = "above the fold" and 17 = "requires scrolling" would result in exactly the same trained model.

In that way, **positions** in LightGBM's API are similar to a categorical feature. Just as with non-ordinal categorical features, an integer representation is just used for memory and computational efficiency... LightGBM does not care about the absolute or relative magnitude of the values.

Unlike a categorical feature, however, **positions** are used to adjust the target to reduce the bias in predictions made by the trained model.

The position file corresponds with training data file line by line, and has one position per line. And if the name of training data file is `train.txt`, the position file should be named as `train.txt.position` and placed in the same folder as the data file. In this case, LightGBM will load the position file automatically if it exists. The positions can also be specified through the `Dataset` constructor when using Python API. If the positions are specified in both approaches, the `.position` file will be ignored.

Currently, implemented is an approach to model position bias by using an idea of Generalized Additive Models (GAM) to linearly decompose the document score s into the sum of a relevance component f and a positional component g : $s(x, pos) = f(x) + g(pos)$ where the former component depends on the original query-document features and the latter depends on the position of an item. During the training, the compound scoring function $s(x, pos)$ is fit with a standard ranking algorithm (e.g., LambdaMART) which boils down to jointly learning the relevance component $f(x)$ (it is later returned as an unbiased model) and the position factors $g(pos)$ that help better explain the observed (biased) labels. Similar score decomposition ideas have previously been applied for classification & pointwise ranking tasks with assumptions of binary labels and binary relevance (a.k.a. "two-tower" models, refer to the papers: [Towards Disentangling Relevance and Bias in Unbiased Learning to Rank](#), [PAL: a position-bias aware learning framework for CTR prediction in live recommender systems](#), [A General Framework for Debiasing in CTR Prediction](#)). In LightGBM, we adapt this idea to general pairwise learning-to-Rank with arbitrary ordinal relevance labels. Besides, GAMs have been used in the context of explainable ML ([Accurate Intelligible Models with Pairwise Interactions](#)) to linearly decompose the contribution of each feature (and possibly their pairwise interactions) to the overall score, for subsequent analysis and interpretation of their effects in the trained models.

LIGHTGBM FAQ

LightGBM Frequently Asked Questions

- *General LightGBM Questions*
- *R-package*
- *Python-package*

Please post questions, feature requests, and bug reports at <https://github.com/microsoft/LightGBM/issues>.

This project is mostly maintained by volunteers, so please be patient. If your request is time-sensitive or more than a month goes by without a response, please tag the maintainers below for help.

- [@guolinke](#) **Guolin Ke**
- [@shiyu1994](#) **Yu Shi**
- [@jameslamb](#) **James Lamb**
- [@jmoralez](#) **José Morales**

13.1 General LightGBM Questions

- 1. *Where do I find more details about LightGBM parameters?*
- 2. *On datasets with millions of features, training does not start (or starts after a very long time).*
- 3. *When running LightGBM on a large dataset, my computer runs out of RAM.*
- 4. *I am using Windows. Should I use Visual Studio or MinGW for compiling LightGBM?*
- 5. *When using LightGBM GPU, I cannot reproduce results over several runs.*
- 6. *Bagging is not reproducible when changing the number of threads.*
- 7. *I tried to use Random Forest mode, and LightGBM crashes!*
- 8. *CPU usage is low (like 10%) in Windows when using LightGBM on very large datasets with many-core systems.*

- 9. When I'm trying to specify a categorical column with the `categorical_feature` parameter, I get the following sequence of warnings, but there are no negative values in the column.
- 10. LightGBM crashes randomly with the error like: `Initializing libomp5.dylib, but found libomp.dylib already initialized.`
- 11. LightGBM hangs when multithreading (OpenMP) and using forking in Linux at the same time.
- 12. Why is early stopping not enabled by default in LightGBM?
- 13. Does LightGBM support direct loading data from zero-based or one-based LibSVM format file?
- 14. Why CMake cannot find the compiler when compiling LightGBM with MinGW?
- 15. Where can I find LightGBM's logo to use it in my presentation?
- 16. LightGBM crashes randomly or operating system hangs during or after running LightGBM.
- 17. Loading LightGBM fails like: `cannot allocate memory in static TLS block`

13.1.1 1. Where do I find more details about LightGBM parameters?

Take a look at Parameters.

13.1.2 2. On datasets with millions of features, training does not start (or starts after a very long time).

Use a smaller value for `bin_construct_sample_cnt` and a larger value for `min_data`.

13.1.3 3. When running LightGBM on a large dataset, my computer runs out of RAM.

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](#)).

13.1.4 4. I am using Windows. Should I use Visual Studio or MinGW for compiling LightGBM?

Visual Studio performs best for LightGBM.

13.1.5 5. When using LightGBM GPU, I cannot reproduce results over several runs.

This is normal and expected behaviour, but you may try to use `gpu_use_dp = true` for reproducibility (see [Microsoft/LightGBM#560](#)). You may also use the CPU version.

13.1.6 6. Bagging is not reproducible when changing the number of threads.

LightGBM bagging is multithreaded, so its output depends on the number of threads used. There is [no workaround](#) currently.

Starting from [#2804](#) bagging result doesn't depend on the number of threads. So this issue should be solved in the latest version.

13.1.7 7. I tried to use Random Forest mode, and LightGBM crashes!

This is expected behaviour for arbitrary parameters. To enable Random Forest, you must use `bagging_fraction` and `feature_fraction` different from 1, along with a `bagging_freq`. [This thread](#) includes an example.

13.1.8 8. CPU usage is low (like 10%) in Windows when using LightGBM on very large datasets with many-core systems.

Please use [Visual Studio](#) as it may be 10x faster than [MinGW](#) especially for very large trees.

13.1.9 9. When I'm trying to specify a categorical column with the `categorical_feature` parameter, I get the following sequence of warnings, but there are no negative values in the column.

```
[LightGBM] [Warning] Met negative value in categorical features, will convert it to NaN
[LightGBM] [Warning] There are no meaningful features, as all feature values are
→ constant.
```

The column you're trying to pass via `categorical_feature` likely contains very large values. Categorical features in LightGBM are limited by `int32` range, so you cannot pass values that are greater than `Int32.MaxValue` (2147483647) as categorical features (see [Microsoft/LightGBM#1359](#)). You should convert them to integers ranging from zero to the number of categories first.

13.1.10 10. LightGBM crashes randomly with the error like: Initializing `libiomp5.dylib`, but found `libomp.dylib` already initialized.

```
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/.
```

Possible Cause: This error means that you have multiple OpenMP libraries installed on your machine and they conflict with each other. (File extensions in the error message may differ depending on the operating system).

If you are using Python distributed by Conda, then it is highly likely that the error is caused by the `numpy` package from Conda which includes the `mkl` package which in turn conflicts with the system-wide library. In this case you can update the `numpy` package in Conda or replace the Conda's OpenMP library instance with system-wide one by creating a symlink to it in Conda environment folder `$CONDA_PREFIX/lib`.

Solution: Assuming you are using macOS with Homebrew, the command which overwrites OpenMP library files in the current active Conda environment with symlinks to the system-wide library ones installed by Homebrew:

```
ln -sf `ls -d "$(brew --cellar libomp)"/*/lib`/* $CONDA_PREFIX/lib
```

The described above fix worked fine before the release of OpenMP 8.0.0 version. Starting from 8.0.0 version, Homebrew formula for OpenMP includes `-DLIBOMP_INSTALL_ALIASES=OFF` option which leads to that the fix doesn't work anymore. However, you can create symlinks to library aliases manually:

```
for LIBOMP_ALIAS in libgomp.dylib libiomp5.dylib libomp.dylib; do sudo ln -sf "$(brew --cellar libomp)"/*/lib/libomp.dylib $CONDA_PREFIX/lib/$LIBOMP_ALIAS; done
```

Another workaround would be removing MKL optimizations from Conda's packages completely:

```
conda install nomkl
```

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

13.1.11 11. LightGBM hangs when multithreading (OpenMP) and using forking in Linux at the same time.

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 (example: using OpenMP for forking), OpenMP will hang inside the forked sessions. Use new processes instead and copy memory as required by creating new processes instead of forking (or, use Intel compilers).

Cloud platform container services may cause LightGBM to hang, if they use Linux fork to run multiple containers on a single instance. For example, LightGBM hangs in AWS Batch array jobs, which [use the ECS agent to manage multiple running jobs](#). Setting `nthreads=1` mitigates the issue.

13.1.12 12. Why is early stopping not enabled by default in LightGBM?

Early stopping involves choosing a validation set, a special type of holdout which is used to evaluate the current state of the model after each iteration to see if training can stop.

In LightGBM, we have decided to require that users specify this set directly. Many options exist for splitting training data into training, test, and validation sets.

The appropriate splitting strategy depends on the task and domain of the data, information that a modeler has but which LightGBM as a general-purpose tool does not.

13.1.13 13. Does LightGBM support direct loading data from zero-based or one-based LibSVM format file?

LightGBM supports loading data from zero-based LibSVM format file directly.

13.1.14 14. Why CMake cannot find the compiler when compiling LightGBM with MinGW?

```
CMake Error: CMAKE_C_COMPILER not set, after EnableLanguage
CMake Error: CMAKE_CXX_COMPILER not set, after EnableLanguage
```

This is a known issue of CMake when using MinGW. The easiest solution is to run again your `cmake` command to bypass the one time stopper from CMake. Or you can upgrade your version of CMake to at least version 3.17.0.

See [Microsoft/LightGBM#3060](#) for more details.

13.1.15 15. Where can I find LightGBM's logo to use it in my presentation?

You can find LightGBM's logo in different file formats and resolutions [here](#).

13.1.16 16. LightGBM crashes randomly or operating system hangs during or after running LightGBM.

Possible Cause: This behavior may indicate that you have multiple OpenMP libraries installed on your machine and they conflict with each other, similarly to the [FAQ #10](#).

If you are using any Python-package that depends on `threadpoolctl`, you also may see the following warning in your logs in this case:

```
/root/miniconda/envs/test-env/lib/python3.8/site-packages/threadpoolctl.py:546: RuntimeWarning:
  Found Intel OpenMP ('libiomp') and LLVM OpenMP ('libomp') loaded at
  the same time. Both libraries are known to be incompatible and this
  can cause random crashes or deadlocks on Linux when loaded in the
  same Python program.

Using threadpoolctl may cause crashes or deadlocks. For more
information and possible workarounds, please see
  https://github.com/joblib/threadpoolctl/blob/master/multiple\_openmp.md
```

Detailed description of conflicts between multiple OpenMP instances is provided in the following document.

Solution: Assuming you are using LightGBM Python-package and conda as a package manager, we strongly recommend using `conda-forge` channel as the only source of all your Python package installations because it contains built-in patches to workaround OpenMP conflicts. Some other workarounds are listed [here](#) under the “Workarounds for Intel OpenMP and LLVM OpenMP case” section.

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

13.1.17 17. Loading LightGBM fails like: cannot allocate memory in static TLS block

When loading LightGBM, you may encounter errors like the following.

```
lib/libgomp.so.1: cannot allocate memory in static TLS block
```

This most commonly happens on aarch64 Linux systems.

gcc's OpenMP library (`libgomp.so`) tries to allocate a small amount of static thread-local storage (“TLS”) when it's dynamically loaded.

That error can happen when the loader isn't able to find a large enough block of memory.

On aarch64 Linux, processes and loaded libraries share the same pool of static TLS, which makes such failures more likely. See these discussions:

- https://bugzilla.redhat.com/show_bug.cgi?id=1722181#c6
- https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100000

If you are experiencing this issue when using the `lightgbm` Python-package, try upgrading to at least v4.6.0.

For older versions of the Python-package, or for other LightGBM APIs, this issue can often be avoided by loading `libgomp.so.1`. That can be done directly by setting environment variable `LD_PRELOAD`, like this:

```
export LD_PRELOAD=/root/miniconda3/envs/test-env/lib/libgomp.so.1
```

It can also be done indirectly by changing the order that other libraries are loaded into processes, which varies by programming language and application type.

For more details, see these discussions:

- <https://github.com/microsoft/LightGBM/pull/6654#issuecomment-2352014275>
- <https://github.com/microsoft/LightGBM/issues/6509>
- <https://maskray.me/blog/2021-02-14-all-about-thread-local-storage>
- https://bugzilla.redhat.com/show_bug.cgi?id=1722181#c6

13.2 R-package

- 1. *Any training command using LightGBM does not work after an error occurred during the training of a previous LightGBM model.*
- 2. *I used setinfo(), tried to print my lgb.Dataset, and now the R console froze!*
- 3. *error in data.table::data.table()...argument 2 is NULL.*
- 4. *package/dependency ‘Matrix’ is not available ...*

13.2.1 1. Any training command using LightGBM does not work after an error occurred during the training of a previous LightGBM model.

In older versions of the R-package (prior to v3.3.0), this could happen occasionally and the solution was to run `lgb.unloader(wipe = TRUE)` to remove all LightGBM-related objects. Some conversation about this could be found in [Microsoft/LightGBM#698](#).

That is no longer necessary as of v3.3.0, and function `lgb.unloader()` has since been removed from the R-package.

13.2.2 2. I used setinfo(), tried to print my lgb.Dataset, and now the R console froze!

As of at least LightGBM v3.3.0, this issue has been resolved and printing a `Dataset` object does not cause the console to freeze.

In older versions, avoid printing the `Dataset` after calling `setinfo()`.

As of LightGBM v4.0.0, `setinfo()` has been replaced by a new method, `set_field()`.

13.2.3 3. error in data.table::data.table()...argument 2 is NULL.

If you are experiencing this error when running `lightgbm`, you may be facing the same issue reported in [#2715](#) and later in [#2989](#). We have seen that in some situations, using `data.table` 1.11.x results in this error. To get around this, you can upgrade your version of `data.table` to at least version 1.12.0.

13.2.4 4. package/dependency ‘Matrix’ is not available ...

In April 2024, `Matrix==1.7-0` was published to CRAN. That version had a floor of R (`>=4.4.0`). `{Matrix}` is a hard runtime dependency of `{lightgbm}`, so on any version of R older than `4.4.0`, running `install.packages("lightgbm")` results in something like the following.

```
package 'Matrix' is not available for this version of R
```

To fix that without upgrading to R 4.4.0 or greater, manually install an older version of `{Matrix}`.

```
install.packages('https://cran.r-project.org/src/contrib/Archive/Matrix/Matrix_1.6-5.tar.  
gz', repos = NULL)
```

13.3 Python-package

- 1. *Error: setup script specifies an absolute path when installing from GitHub using `python setup.py install`.*
- 2. *Error messages: Cannot ... before construct dataset.*
- 3. *I encounter segmentation faults (segfaults) randomly after installing LightGBM from PyPI using `pip install lightgbm`.*
- 4. *I would like to install LightGBM from conda. What channel should I choose?*
- 5. *How do I subclass scikit-learn estimators?*

13.3.1 1. Error: setup script specifies an absolute path when installing from GitHub using `python setup.py install`.

Note

As of v4.0.0, `lightgbm` does not support directly invoking `setup.py`. This answer refers only to versions of `lightgbm` prior to v4.0.0.

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

This error should be solved in latest version. If you still meet this error, try to remove `lightgbm.egg-info` folder in your Python-package and reinstall, or check [this thread on stackoverflow](#).

13.3.2 2. Error messages: Cannot ... before construct dataset.

I see error messages like...

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

but I've already constructed a dataset by some code like:

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

or error messages like

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

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

- get label (or weight/init_score/group/data) before constructing a dataset, it's same as get `self.label`;
- set label (or weight/init_score/group) before constructing a dataset, it's same as `self.label=some_label_array`;
- get num_data (or num_feature) before constructing a dataset, you can get data with `self.data`. Then, if your data is `numpy.ndarray`, use some code like `self.data.shape`. But do not do this after subsetting the Dataset, because you'll get always `None`;
- set predictor (or reference/categorical feature) after constructing a dataset, you should set `free_raw_data=False` or init a Dataset object with the same raw data.

13.3.3 3. I encounter segmentation faults (segfaults) randomly after installing LightGBM from PyPI using `pip install lightgbm`.

We are doing our best to provide universal wheels which have high running speed and are compatible with any hardware, OS, compiler, etc. at the same time. However, sometimes it's just impossible to guarantee the possibility of usage of LightGBM in any specific environment (see [Microsoft/LightGBM#1743](#)).

Therefore, the first thing you should try in case of segfaults is **compiling from the source** using `pip install --no-binary lightgbm lightgbm`. For the OS-specific prerequisites see <https://github.com/microsoft/LightGBM/blob/master/python-package/README.rst>.

Also, feel free to post a new issue in our GitHub repository. We always look at each case individually and try to find a root cause.

13.3.4 4. I would like to install LightGBM from conda. What channel should I choose?

We strongly recommend installation from the `conda-forge` channel and not from the `default` one.

For some specific examples, see [this comment](#).

In addition, as of `lightgbm==4.4.0`, the `conda-forge` package automatically supports CUDA-based GPU acceleration.

13.3.5 5. How do I subclass scikit-learn estimators?

For `lightgbm <= 4.5.0`, copy all of the constructor arguments from the corresponding `lightgbm` class into the constructor of your custom estimator.

For later versions, just ensure that the constructor of your custom estimator calls `super().__init__()`.

Consider the example below, which implements a regressor that allows creation of truncated predictions. This pattern will work with `lightgbm > 4.5.0`.

```
import numpy as np
from lightgbm import LGBMRegressor
from sklearn.datasets import make_regression

class TruncatedRegressor(LGBMRegressor):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def predict(self, X, max_score: float = np.inf):
        preds = super().predict(X)
        np.clip(preds, a_min=None, a_max=max_score, out=preds)
        return preds

X, y = make_regression(n_samples=1_000, n_features=4)

reg_trunc = TruncatedRegressor().fit(X, y)

preds = reg_trunc.predict(X)
print(f'mean: {preds.mean():.2f}, max: {preds.max():.2f}')
# mean: -6.81, max: 345.10

preds_trunc = reg_trunc.predict(X, max_score=preds.mean())
print(f'mean: {preds_trunc.mean():.2f}, max: {preds_trunc.max():.2f}')
# mean: -56.50, max: -6.81
```


DEVELOPMENT GUIDE

14.1 Algorithms

Refer to [Features](#) for understanding of important algorithms used in LightGBM.

14.2 Classes and Code Structure

14.2.1 Important Classes

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, etc.)
Config	Stores parameters and configurations
Dataset	Stores information of dataset
DatasetLoader	Used to construct dataset
FeatureGroup	Stores the data of feature, could be multiple features
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

14.2.2 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 related 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

14.3 Documents API

Refer to [docs](#) README.

14.4 C API

Refer to [C API](#) or the comments in `c_api.h` file, from which the documentation is generated.

14.5 Tests

C++ unit tests are located in the `./tests/cpp_tests` folder and written with the help of Google Test framework. To run tests locally first refer to the [Installation Guide](#) for how to build tests and then simply run compiled executable file. It is highly recommended to build tests with [sanitizers](#).

14.6 High Level Language Package

See the implementations at [Python-package](#) and [R-package](#).

14.7 Questions

Refer to [FAQ](#).

Also feel free to open [issues](#) if you met problems.

GPU TUNING GUIDE AND PERFORMANCE COMPARISON

15.1 How It Works?

In LightGBM, the main computation cost during training is building the feature histograms. We use an efficient algorithm on GPU to accelerate this process. The implementation is highly modular, and works for all learning tasks (classification, ranking, regression, etc). GPU acceleration also works in distributed learning settings. GPU algorithm implementation is based on OpenCL and can work with a wide range of GPUs.

15.2 Supported Hardware

We target AMD Graphics Core Next (GCN) architecture and NVIDIA Maxwell and Pascal architectures. Most AMD GPUs released after 2012 and NVIDIA GPUs released after 2014 should be supported. We have tested the GPU implementation on the following GPUs:

- AMD RX 480 with AMDGPU-pro driver 16.60 on Ubuntu 16.10
- AMD R9 280X (aka Radeon HD 7970) with fglrx driver 15.302.2301 on Ubuntu 16.10
- NVIDIA GTX 1080 with driver 375.39 and CUDA 8.0 on Ubuntu 16.10
- NVIDIA Titan X (Pascal) with driver 367.48 and CUDA 8.0 on Ubuntu 16.04
- NVIDIA Tesla M40 with driver 375.39 and CUDA 7.5 on Ubuntu 16.04

Using the following hardware is discouraged:

- NVIDIA Kepler (K80, K40, K20, most GeForce GTX 700 series GPUs) or earlier NVIDIA GPUs. They don't support hardware atomic operations in local memory space and thus histogram construction will be slow.
- AMD VLIW4-based GPUs, including Radeon HD 6xxx series and earlier GPUs. These GPUs have been discontinued for years and are rarely seen nowadays.

15.3 How to Achieve Good Speedup on GPU

1. You want to run a few datasets that we have verified with good speedup (including Higgs, epsilon, Bosch, etc) to ensure your setup is correct. If you have multiple GPUs, make sure to set `gpu_platform_id` and `gpu_device_id` to use the desired GPU. Also make sure your system is idle (especially when using a shared computer) to get accuracy performance measurements.
2. GPU works best on large scale and dense datasets. If dataset is too small, computing it on GPU is inefficient as the data transfer overhead can be significant. If you have categorical features, use the `categorical_column` option and input them into LightGBM directly; do not convert them into one-hot variables.
3. To get good speedup with GPU, it is suggested to use a smaller number of bins. Setting `max_bin=63` is recommended, as it usually does not noticeably affect training accuracy on large datasets, but GPU training can

be significantly faster than using the default bin size of 255. For some dataset, even using 15 bins is enough (`max_bin=15`); using 15 bins will maximize GPU performance. Make sure to check the run log and verify that the desired number of bins is used.

4. Try to use single precision training (`gpu_use_dp=false`) when possible, because most GPUs (especially NVIDIA consumer GPUs) have poor double-precision performance.

15.4 Performance Comparison

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

Data	Task	Link	#Exam-ples	#Fea-tures	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
device = gpu
gpu_platform_id = 0
```

(continues on next page)

(continued from previous page)

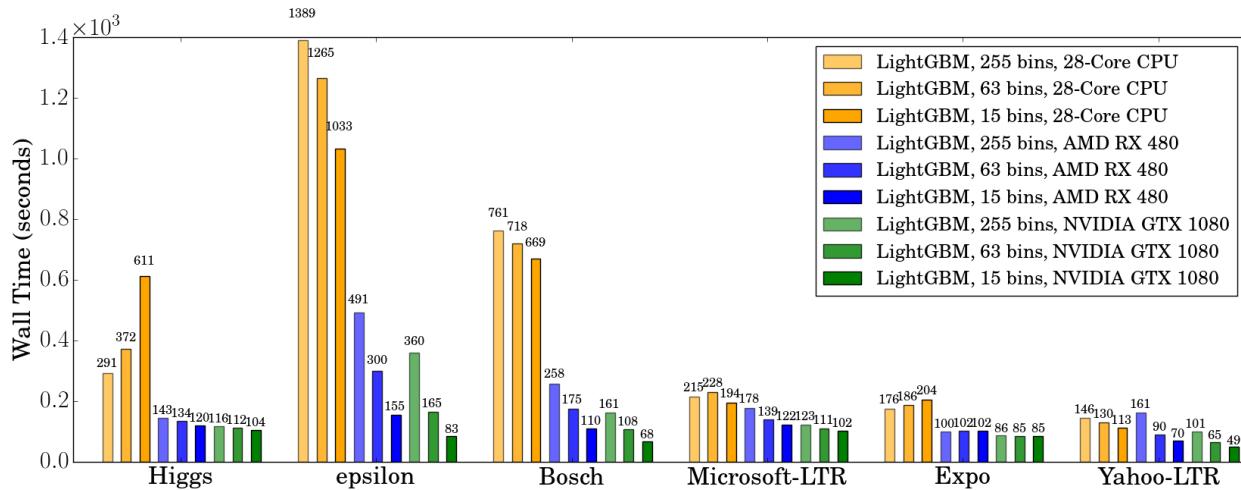
```
gpu_device_id = 0
num_thread = 28
```

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 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 bins	255	CPU bins	63	CPU bins	15	GPU bins	255	GPU bins	63	GPU bins	15
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	0.730824		0.730165		0.729647		0.730936		0.732257		0.73114	
NDCG ₁												
Yahoo-LTR	0.738687		0.737243		0.736445		0.73698		0.739474		0.735868	
NDCG ₃												
Yahoo-LTR	0.756609		0.755729		0.754607		0.756206		0.757007		0.754203	
NDCG ₅												
Yahoo-LTR	0.79655		0.795827		0.795273		0.795894		0.797302		0.795584	
NDCG ₁₀												
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.

15.5 Memory Usage

The next table shows GPU memory usage reported by `nvidia-smi` during training with 63 bins. We can see that even the largest dataset just uses about 1 GB of GPU memory, indicating that our GPU implementation can scale to huge datasets over 10x larger than Bosch or Epsilon. Also, we can observe that generally a larger dataset (using more GPU memory, like Epsilon or Bosch) has better speedup, because the overhead of invoking GPU functions becomes significant when the dataset is small.

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

15.6 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.

GPU SDK CORRESPONDENCE AND DEVICE TARGETING TABLE

16.1 GPU Targets Table

OpenCL is a universal massively parallel programming framework that targets 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
PoCL	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 from our GitHub repo ([for Linux](#) and [for Windows](#)).

16.2 Query OpenCL Devices in Your System

Your system might have multiple GPUs from different vendors (“platforms”) installed. Setting up LightGBM GPU device requires two parameters: [OpenCL Platform ID](#) (`gpu_platform_id`) and [OpenCL Device ID](#) (`gpu_device_id`). Generally speaking, each vendor provides an OpenCL platform, and devices from the same vendor have different device IDs under that platform. For example, if your system has an Intel integrated GPU and two discrete GPUs from AMD, you will have two OpenCL platforms (with `gpu_platform_id=0` and `gpu_platform_id=1`). If the platform 0 is Intel, it has one device (`gpu_device_id=0`) representing the Intel GPU; if the platform 1 is AMD, it has two devices (`gpu_device_id=0`, `gpu_device_id=1`) representing the two AMD GPUs. If you have a discrete GPU by AMD/NVIDIA and an integrated GPU by Intel, make sure to select the correct `gpu_platform_id` to use the discrete GPU as it usually provides better performance.

On Windows, OpenCL devices can be queried using [GPUCapsViewer](#), under the OpenCL tab. Note that the platform and device IDs reported by this utility start from 1. So you should minus the reported IDs by 1.

On Linux, OpenCL devices can be listed using the `clinfo` command. On Ubuntu, you can install `clinfo` by executing `sudo apt-get install clinfo`.

16.3 Examples

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

```
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,
+                 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)
```

(continues on next page)

(continued from previous page)

```
[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 depth=8
[1]: test's rmse:1.10643e-17
[LightGBM] [Info] No further splits with positive gain, best gain: -inf
[LightGBM] [Info] Trained a tree with leaves=7 and depth=5
[2]: test's rmse:0
```

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
[LightGBM] [Info] 40 dense feature groups (0.12 MB) transferred to GPU in 0.004540 secs. ↵
→ 76 sparse feature groups.
[LightGBM] [Info] No further splits with positive gain, best gain: -inf
[LightGBM] [Info] Trained a tree with leaves=16 and depth=8
[1]: test's rmse:1.10643e-17
```

(continues on next page)

(continued from previous page)

```
[LightGBM] [Info] No further splits with positive gain, best gain: -inf
[LightGBM] [Info] Trained a tree with leaves=7 and depth=5
[2]:    test's rmse:0
```

Known issues:

- Using a bad combination of `gpu_platform_id` and `gpu_device_id` can potentially lead to a **crash** due to OpenCL driver issues on some machines (you will lose your entire session content). Beware of it.
- On some systems, if you have integrated graphics card (Intel HD Graphics) and a dedicated graphics card (AMD, NVIDIA), the dedicated graphics card will automatically override the integrated graphics card. The workaround is to disable your dedicated graphics card to be able to use your integrated graphics card.

The content of this document was very outdated and is no longer available to avoid any misleadings.

Starting from the 3.2.0 version LightGBM Python packages have been having built-in support of training on GPU devices.

The content of this document was very outdated and is no longer available to avoid any misleadings.

DOCUMENTATION

Documentation for LightGBM is generated using [Sphinx](#) and [Breathe](#), which works on top of [Doxygen](#) output.

List of parameters and their descriptions in `Parameters.rst` is generated automatically from comments in `config` file by this script.

After each commit on `master`, documentation is updated and published to [Read the Docs](#).

17.1 Build

It is not necessary to re-build this documentation while modifying LightGBM's source code. The HTML files generated using [Sphinx](#) are not checked into source control. However, you may want to build them locally during development to test changes.

17.1.1 Docker

The most reliable way to build the documentation locally is with Docker, using the same images [Read the Docs](#) uses.

Run the following from the root of this repository to pull the relevant image and run a container locally.

```
docker run \
  --rm \
  --user=0 \
  -v $(pwd):/opt/LightGBM \
  --env C_API=true \
  --env CONDA=/opt/miniforge \
  --env READTHEDOCS=true \
  --workdir=/opt/LightGBM/docs \
  --entrypoint="" \
  readthedocs/build:ubuntu-24.04-2024.06.17 \
  /bin/bash build-docs.sh
```

When that code completes, open `docs/_build/html/index.html` in your browser.

Note

The navigation in these locally-built docs does not link to the local copy of the R documentation. To view the local version of the R docs, open `docs/_build/html/R/index.html` in your browser.

17.1.2 Without Docker

You can build the documentation locally without Docker. Just install Doxygen and run in docs folder

```
pip install breathe sphinx 'sphinx_rtd_theme>=0.5'  
make html
```

Note that this will not build the R documentation. Consider using common R utilities for documentation generation, if you need it. Or use the Docker-based approach described above to build the R documentation locally.

Optionally, you may also install `scikit-learn` and get richer documentation for the classes in `Scikit-learn API`.

If you faced any problems with Doxygen installation or you simply do not need documentation for C code, it is possible to build the documentation without it:

```
pip install sphinx 'sphinx_rtd_theme>=0.5'  
export C_API=NO || set C_API=NO  
make html
```

CHAPTER
EIGHTEEN

INDICES AND TABLES

- genindex

INDEX

Symbols

`__init__()` (`lightgbm.Booster` method), 113
`__init__()` (`lightgbm.CVBooster` method), 125
`__init__()` (`lightgbm.DaskLGBMClassifier` method), 180
`__init__()` (`lightgbm.DaskLGBMRanker` method), 205
`__init__()` (`lightgbm.DaskLGBMRegressor` method), 193
`__init__()` (`lightgbm.Dataset` method), 105
`__init__()` (`lightgbm.LGBMClassifier` method), 143
`__init__()` (`lightgbm.LGBMModel` method), 131
`__init__()` (`lightgbm.LGBMRanker` method), 168
`__init__()` (`lightgbm.LGBMRegressor` method), 156
`__init__()` (`lightgbm.Sequence` method), 127

A

`add_features_from()` (`lightgbm.Dataset` method), 107
`add_valid()` (`lightgbm.Booster` method), 114

B

`batch_size` (`lightgbm.Sequence` attribute), 127
`best_iteration` (`lightgbm.CVBooster` attribute), 125
`best_iteration_` (`lightgbm.DaskLGBMClassifier` property), 183
`best_iteration_` (`lightgbm.DaskLGBMRanker` property), 208
`best_iteration_` (`lightgbm.DaskLGBMRegressor` property), 197
`best_iteration_` (`lightgbm.LGBMClassifier` property), 146
`best_iteration_` (`lightgbm.LGBMModel` property), 135
`best_iteration_` (`lightgbm.LGBMRanker` property), 172
`best_iteration_` (`lightgbm.LGBMRegressor` property), 160
`best_score_` (`lightgbm.DaskLGBMClassifier` property), 183
`best_score_` (`lightgbm.DaskLGBMRanker` property), 208
`best_score_` (`lightgbm.DaskLGBMRegressor` property), 197

`best_score_` (`lightgbm.LGBMClassifier` property), 146
`best_score_` (`lightgbm.LGBMModel` property), 135
`best_score_` (`lightgbm.LGBMRanker` property), 172
`best_score_` (`lightgbm.LGBMRegressor` property), 160
`Booster` (class in `lightgbm`), 113
`booster_` (`lightgbm.DaskLGBMClassifier` property), 183
`booster_` (`lightgbm.DaskLGBMRanker` property), 209
`booster_` (`lightgbm.DaskLGBMRegressor` property), 197
`booster_` (`lightgbm.LGBMClassifier` property), 146
`booster_` (`lightgbm.LGBMModel` property), 135
`booster_` (`lightgbm.LGBMRanker` property), 172
`booster_` (`lightgbm.LGBMRegressor` property), 160
`BoosterHandle` (C type), 70
`boosters` (`lightgbm.CVBooster` attribute), 125
`ByteBufferHandle` (C type), 70

C

`C_API_DTYPE_FLOAT32` (C macro), 69
`C_API_DTYPE_FLOAT64` (C macro), 69
`C_API_DTYPE_INT32` (C macro), 69
`C_API_DTYPE_INT64` (C macro), 69
`C_API_FEATURE_IMPORTANCE_GAIN` (C macro), 69
`C_API_FEATURE_IMPORTANCE_SPLIT` (C macro), 69
`C_API_MATRIX_TYPE_CSC` (C macro), 69
`C_API_MATRIX_TYPE_CSR` (C macro), 70
`C_API_PREDICT_CONTRIB` (C macro), 70
`C_API_PREDICT_LEAF_INDEX` (C macro), 70
`C_API_PREDICT_NORMAL` (C macro), 70
`C_API_PREDICT_RAW_SCORE` (C macro), 70
`classes_` (`lightgbm.DaskLGBMClassifier` property), 184
`classes_` (`lightgbm.LGBMClassifier` property), 146
`client_` (`lightgbm.DaskLGBMClassifier` property), 184
`client_` (`lightgbm.DaskLGBMRanker` property), 209
`client_` (`lightgbm.DaskLGBMRegressor` property), 197
`construct()` (`lightgbm.Dataset` method), 107
`create_tree_digraph()` (in module `lightgbm`), 222
`create_valid()` (`lightgbm.Dataset` method), 107
`current_iteration()` (`lightgbm.Booster` method), 114
`cv()` (in module `lightgbm`), 129

`CVBooster` (*class in lightgbm*), 125

D

`DaskLGBMClassifier` (*class in lightgbm*), 180
`DaskLGBMRanker` (*class in lightgbm*), 205
`DaskLGBMRegressor` (*class in lightgbm*), 193
`Dataset` (*class in lightgbm*), 105
`DatasetHandle` (*C type*), 70
`dump_model()` (*lightgbm.Booster method*), 114

E

`early_stopping()` (*in module lightgbm*), 216
`eval()` (*lightgbm.Booster method*), 115
`eval_train()` (*lightgbm.Booster method*), 115
`eval_valid()` (*lightgbm.Booster method*), 116
`evals_result_` (*lightgbm.DaskLGBMClassifier property*), 184
`evals_result_` (*lightgbm.DaskLGBMRanker property*), 209
`evals_result_` (*lightgbm.DaskLGBMRegressor property*), 197
`evals_result_` (*lightgbm.LGBMClassifier property*), 146
`evals_result_` (*lightgbm.LGBMModel property*), 135
`evals_result_` (*lightgbm.LGBMRanker property*), 172
`evals_result_` (*lightgbm.LGBMRegressor property*), 160

F

`FastConfigHandle` (*C type*), 70
`feature_importance()` (*lightgbm.Booster method*), 117
`feature_importances_` (*lightgbm.DaskLGBMClassifier property*), 184
`feature_importances_` (*lightgbm.DaskLGBMRanker property*), 209
`feature_importances_` (*lightgbm.DaskLGBMRegressor property*), 197
`feature_importances_` (*lightgbm.LGBMClassifier property*), 147
`feature_importances_` (*lightgbm.LGBMModel property*), 135
`feature_importances_` (*lightgbm.LGBMRanker property*), 172
`feature_importances_` (*lightgbm.LGBMRegressor property*), 160
`feature_name()` (*lightgbm.Booster method*), 117
`feature_name_` (*lightgbm.DaskLGBMClassifier property*), 184
`feature_name_` (*lightgbm.DaskLGBMRanker property*), 209
`feature_name_` (*lightgbm.DaskLGBMRegressor property*), 198

`feature_name_` (*lightgbm.LGBMClassifier property*), 147
`feature_name_` (*lightgbm.LGBMModel property*), 135
`feature_name_` (*lightgbm.LGBMRanker property*), 172
`feature_name_` (*lightgbm.LGBMRegressor property*), 160
`feature_names_in_` (*lightgbm.DaskLGBMClassifier property*), 184
`feature_names_in_` (*lightgbm.DaskLGBMRanker property*), 209
`feature_names_in_` (*lightgbm.DaskLGBMRegressor property*), 198
`feature_names_in_` (*lightgbm.LGBMClassifier property*), 147
`feature_names_in_` (*lightgbm.LGBMModel property*), 136
`feature_names_in_` (*lightgbm.LGBMRanker property*), 173
`feature_names_in_` (*lightgbm.LGBMRegressor property*), 161
`feature_num_bin()` (*lightgbm.Dataset method*), 108
`fit()` (*lightgbm.DaskLGBMClassifier method*), 184
`fit()` (*lightgbm.DaskLGBMRanker method*), 209
`fit()` (*lightgbm.DaskLGBMRegressor method*), 198
`fit()` (*lightgbm.LGBMClassifier method*), 147
`fit()` (*lightgbm.LGBMModel method*), 136
`fit()` (*lightgbm.LGBMRanker method*), 173
`fit()` (*lightgbm.LGBMRegressor method*), 161
`free_dataset()` (*lightgbm.Booster method*), 117
`free_network()` (*lightgbm.Booster method*), 117

G

`get_data()` (*lightgbm.Dataset method*), 108
`get_feature_name()` (*lightgbm.Dataset method*), 108
`get_field()` (*lightgbm.Dataset method*), 108
`get_group()` (*lightgbm.Dataset method*), 109
`get_init_score()` (*lightgbm.Dataset method*), 109
`get_label()` (*lightgbm.Dataset method*), 109
`get_leaf_output()` (*lightgbm.Booster method*), 117
`get_metadata_routing()` (*lightgbm.DaskLGBMClassifier method*), 186
`get_metadata_routing()` (*lightgbm.DaskLGBMRanker method*), 211
`get_metadata_routing()` (*lightgbm.DaskLGBMRegressor method*), 200
`get_metadata_routing()` (*lightgbm.LGBMClassifier method*), 149
`get_metadata_routing()` (*lightgbm.LGBMModel method*), 138
`get_metadata_routing()` (*lightgbm.LGBMRanker method*), 175
`get_metadata_routing()` (*lightgbm.LGBMRegressor method*), 163

get_params() (*lightgbm.DaskLGBMClassifier* method), 186
 get_params() (*lightgbm.DaskLGBMRanker* method), 211
 get_params() (*lightgbm.DaskLGBMRegressor* method), 200
 get_params() (*lightgbm.Dataset* method), 109
 get_params() (*lightgbm.LGBMClassifier* method), 149
 get_params() (*lightgbm.LGBMModel* method), 138
 get_params() (*lightgbm.LGBMRanker* method), 175
 get_params() (*lightgbm.LGBMRegressor* method), 163
 get_position() (*lightgbm.Dataset* method), 109
 get_ref_chain() (*lightgbm.Dataset* method), 110
 get_split_value_histogram() (*lightgbm.Booster* method), 117
 get_weight() (*lightgbm.Dataset* method), 110

|

INLINE_FUNCTION (*C macro*), 70

L

LastErrorMsg (*C function*), 70
 LGBM_BoosterAddValidData (*C function*), 70
 LGBM_BoosterCalcNumPredict (*C function*), 71
 LGBM_BoosterCreate (*C function*), 71
 LGBM_BoosterCreateFromModelfile (*C function*), 71
 LGBM_BoosterDumpModel (*C function*), 72
 LGBM_BoosterFeatureImportance (*C function*), 72
 LGBM_BoosterFree (*C function*), 72
 LGBM_BoosterFreePredictSparse (*C function*), 73
 LGBM_BoosterGetCurrentIteration (*C function*), 73
 LGBM_BoosterGetEval (*C function*), 73
 LGBM_BoosterGetEvalCounts (*C function*), 73
 LGBM_BoosterGetEvalNames (*C function*), 74
 LGBM_BoosterGetFeatureNames (*C function*), 74
 LGBM_BoosterGetLeafValue (*C function*), 74
 LGBM_BoosterGetLinear (*C function*), 75
 LGBM_BoosterGetLoadedParam (*C function*), 75
 LGBM_BoosterGetLowerBoundValue (*C function*), 75
 LGBM_BoosterGetNumClasses (*C function*), 75
 LGBM_BoosterGetNumFeature (*C function*), 75
 LGBM_BoosterGetNumPredict (*C function*), 76
 LGBM_BoosterGetPredict (*C function*), 76
 LGBM_BoosterGetUpperBoundValue (*C function*), 76
 LGBM_BoosterLoadModelFromString (*C function*), 76
 LGBM_BoosterMerge (*C function*), 77
 LGBM_BoosterNumberOfTotalModel (*C function*), 77
 LGBM_BoosterNumModelPerIteration (*C function*), 77
 LGBM_BoosterPredictForArrow (*C function*), 77
 LGBM_BoosterPredictForCSC (*C function*), 78
 LGBM_BoosterPredictForCSR (*C function*), 79
 LGBM_BoosterPredictForCSRSingleRow (*C function*), 80
 LGBM_BoosterPredictForCSRSingleRowFast (*C function*), 81
 LGBM_BoosterPredictForCSRSingleRowFastInit (*C function*), 82
 LGBM_BoosterPredictForFile (*C function*), 83
 LGBM_BoosterPredictForMat (*C function*), 83
 LGBM_BoosterPredictForMats (*C function*), 84
 LGBM_BoosterPredictForMatSingleRow (*C function*), 85
 LGBM_BoosterPredictForMatSingleRowFast (*C function*), 86
 LGBM_BoosterPredictForMatSingleRowFastInit (*C function*), 86
 LGBM_BoosterPredictSparseOutput (*C function*), 87
 LGBM_BoosterRefit (*C function*), 88
 LGBM_BoosterResetParameter (*C function*), 88
 LGBM_BoosterResetTrainingData (*C function*), 88
 LGBM_BoosterRollbackOneIter (*C function*), 88
 LGBM_BoosterSaveModel (*C function*), 89
 LGBM_BoosterSaveModelToString (*C function*), 89
 LGBM_BoosterSetLeafValue (*C function*), 89
 LGBM_BoosterShuffleModels (*C function*), 90
 LGBM_BoosterUpdateOneIter (*C function*), 90
 LGBM_BoosterUpdateOneIterCustom (*C function*), 90
 LGBM_BoosterValidateFeatureNames (*C function*), 91
 LGBM_ByteBufferFree (*C function*), 91
 LGBM_ByteBufferGetAt (*C function*), 91
 LGBM_DatasetAddFeaturesFrom (*C function*), 91
 LGBM_DatasetCreateByReference (*C function*), 91
 LGBM_DatasetCreateFromArrow (*C function*), 92
 LGBM_DatasetCreateFromCSC (*C function*), 92
 LGBM_DatasetCreateFromCSR (*C function*), 92
 LGBM_DatasetCreateFromCSRFunc (*C function*), 93
 LGBM_DatasetCreateFromFile (*C function*), 93
 LGBM_DatasetCreateFromMat (*C function*), 94
 LGBM_DatasetCreateFromMats (*C function*), 94
 LGBM_DatasetCreateFromSampledColumn (*C function*), 95
 LGBM_DatasetCreateFromSerializedReference (*C function*), 95
 LGBM_DatasetDumpText (*C function*), 95
 LGBM_DatasetFree (*C function*), 96
 LGBM_DatasetGetFeatureNames (*C function*), 96
 LGBM_DatasetGetFeatureNumBin (*C function*), 96
 LGBM_DatasetGetField (*C function*), 96
 LGBM_DatasetGetNumData (*C function*), 97
 LGBM_DatasetGetNumFeature (*C function*), 97
 LGBM_DatasetGetSubset (*C function*), 97
 LGBM_DatasetInitStreaming (*C function*), 97
 LGBM_DatasetMarkFinished (*C function*), 98
 LGBM_DatasetPushRows (*C function*), 98
 LGBM_DatasetPushRowsByCSR (*C function*), 98

LGBM_DatasetPushRowsByCSRWithMetadata (*C function*), 99
LGBM_DatasetPushRowsWithMetadata (*C function*), 99
LGBM_DatasetSaveBinary (*C function*), 100
LGBM_DatasetSerializeReferenceToBinary (*C function*), 100
LGBM_DatasetSetFeatureNames (*C function*), 100
LGBM_DatasetSetField (*C function*), 101
LGBM_DatasetSetFieldFromArrow (*C function*), 101
LGBM_DatasetSetWaitForManualFinish (*C function*), 102
LGBM_DatasetUpdateParamChecking (*C function*), 102
LGBM_DumpParamAliases (*C function*), 102
LGBM_FastConfigFree (*C function*), 102
LGBM_GetLastError (*C function*), 102
LGBM_GetMaxThreads (*C function*), 102
LGBM_GetSampleCount (*C function*), 103
LGBM_NetworkFree (*C function*), 103
LGBM_NetworkInit (*C function*), 103
LGBM_NetworkInitWithFunctions (*C function*), 103
LGBM_RegisterLogCallback (*C function*), 104
LGBM_SampleIndices (*C function*), 104
LGBM_SetLastError (*C function*), 104
LGBM_SetMaxThreads (*C function*), 104
LGBMClassifier (*class in lightgbm*), 142
LGBMModel (*class in lightgbm*), 131
LGBMRanker (*class in lightgbm*), 168
LGBMRegressor (*class in lightgbm*), 156
log_evaluation() (*in module lightgbm*), 217
lower_bound() (*lightgbm.Booster method*), 118

M

model_from_string() (*lightgbm.Booster method*), 118
model_from_string() (*lightgbm.CVBooster method*), 126
model_to_string() (*lightgbm.Booster method*), 118
model_to_string() (*lightgbm.CVBooster method*), 126

N

n_classes_ (*lightgbm.DaskLGBMClassifier property*), 187
n_classes_ (*lightgbm.LGBMClassifier property*), 149
n_estimators_ (*lightgbm.DaskLGBMClassifier property*), 187
n_estimators_ (*lightgbm.DaskLGBMRanker property*), 212
n_estimators_ (*lightgbm.DaskLGBMRegressor property*), 200
n_estimators_ (*lightgbm.LGBMClassifier property*), 150
n_estimators_ (*lightgbm.LGBMModel property*), 138

O

objective_ (*lightgbm.DaskLGBMClassifier property*), 187
objective_ (*lightgbm.DaskLGBMRanker property*), 212
objective_ (*lightgbm.DaskLGBMRegressor property*), 200
objective_ (*lightgbm.LGBMClassifier property*), 150
objective_ (*lightgbm.LGBMModel property*), 139
objective_ (*lightgbm.LGBMRanker property*), 176
objective_ (*lightgbm.LGBMRegressor property*), 164

P

plot_importance() (*in module lightgbm*), 218

`plot_metric()` (in module `lightgbm`), 220
`plot_split_value_histogram()` (in module `lightgbm`), 219
`plot_tree()` (in module `lightgbm`), 221
`predict()` (`lightgbm.Booster` method), 119
`predict()` (`lightgbm.DaskLGBMClassifier` method), 187
`predict()` (`lightgbm.DaskLGBMRanker` method), 212
`predict()` (`lightgbm.DaskLGBMRegressor` method), 201
`predict()` (`lightgbm.LGBMClassifier` method), 150
`predict()` (`lightgbm.LGBMModel` method), 139
`predict()` (`lightgbm.LGBMRanker` method), 176
`predict()` (`lightgbm.LGBMRegressor` method), 164
`predict_proba()` (`lightgbm.DaskLGBMClassifier` method), 188
`predict_proba()` (`lightgbm.LGBMClassifier` method), 151

R

`record_evaluation()` (in module `lightgbm`), 217
`refit()` (`lightgbm.Booster` method), 120
`register_logger()` (in module `lightgbm`), 224
`reset_parameter()` (in module `lightgbm`), 218
`reset_parameter()` (`lightgbm.Booster` method), 122
`rollback_one_iter()` (`lightgbm.Booster` method), 122

S

`save_binary()` (`lightgbm.Dataset` method), 110
`save_model()` (`lightgbm.Booster` method), 122
`save_model()` (`lightgbm.CVBooster` method), 126
`score()` (`lightgbm.DaskLGBMClassifier` method), 189
`score()` (`lightgbm.DaskLGBMRegressor` method), 201
`score()` (`lightgbm.LGBMClassifier` method), 152
`score()` (`lightgbm.LGBMRegressor` method), 165
`Sequence` (class in `lightgbm`), 127
`set_categorical_feature()` (`lightgbm.Dataset` method), 111
`set_feature_name()` (`lightgbm.Dataset` method), 111
`set_field()` (`lightgbm.Dataset` method), 111
`set_fit_request()` (`lightgbm.DaskLGBMClassifier` method), 189
`set_fit_request()` (`lightgbm.DaskLGBMRanker` method), 213
`set_fit_request()` (`lightgbm.DaskLGBMRegressor` method), 202
`set_fit_request()` (`lightgbm.LGBMClassifier` method), 152
`set_fit_request()` (`lightgbm.LGBMModel` method), 140
`set_fit_request()` (`lightgbm.LGBMRanker` method), 177
`set_fit_request()` (`lightgbm.LGBMRegressor` method), 165

`set_group()` (`lightgbm.Dataset` method), 111
`set_init_score()` (`lightgbm.Dataset` method), 111
`set_label()` (`lightgbm.Dataset` method), 112
`set_leaf_output()` (`lightgbm.Booster` method), 122
`set_network()` (`lightgbm.Booster` method), 123
`set_params()` (`lightgbm.DaskLGBMClassifier` method), 191
`set_params()` (`lightgbm.DaskLGBMRanker` method), 214
`set_params()` (`lightgbm.DaskLGBMRegressor` method), 203
`set_params()` (`lightgbm.LGBMClassifier` method), 154
`set_params()` (`lightgbm.LGBMModel` method), 141
`set_params()` (`lightgbm.LGBMRanker` method), 178
`set_params()` (`lightgbm.LGBMRegressor` method), 166
`set_position()` (`lightgbm.Dataset` method), 112
`set_predict_proba_request()` (`lightgbm.DaskLGBMClassifier` method), 191
`set_predict_proba_request()` (`lightgbm.LGBMClassifier` method), 154
`set_predict_request()` (`lightgbm.DaskLGBMClassifier` method), 192
`set_predict_request()` (`lightgbm.DaskLGBMRanker` method), 215
`set_predict_request()` (`lightgbm.DaskLGBMRegressor` method), 203
`set_predict_request()` (`lightgbm.LGBMClassifier` method), 155
`set_predict_request()` (`lightgbm.LGBMModel` method), 141
`set_predict_request()` (`lightgbm.LGBMRanker` method), 178
`set_predict_request()` (`lightgbm.LGBMRegressor` method), 167
`set_reference()` (`lightgbm.Dataset` method), 112
`set_score_request()` (`lightgbm.DaskLGBMClassifier` method), 193
`set_score_request()` (`lightgbm.DaskLGBMRegressor` method), 204
`set_score_request()` (`lightgbm.LGBMClassifier` method), 156
`set_score_request()` (`lightgbm.LGBMRegressor` method), 168
`set_train_data_name()` (`lightgbm.Booster` method), 123
`set_weight()` (`lightgbm.Dataset` method), 112
`shuffle_models()` (`lightgbm.Booster` method), 123
`subset()` (`lightgbm.Dataset` method), 113

T

`THREAD_LOCAL` (C macro), 70
`to_local()` (`lightgbm.DaskLGBMClassifier` method), 193
`to_local()` (`lightgbm.DaskLGBMRanker` method), 215

`to_local()` (*lightgbm.DaskLGBMRegressor* method),
205
`train()` (*in module lightgbm*), 128
`trees_to_dataframe()` (*lightgbm.Booster* method),
123

U

`update()` (*lightgbm.Booster* method), 124
`upper_bound()` (*lightgbm.Booster* method), 125