

# LightGBM使用

## 一、安装

### 1. 安装步骤：

- 下载源码，编译 `lib_lightgbm.so`（对于 `windows` 则是 `lib_lightgbm.dll`）

```
git clone --recursive https://github.com/Microsoft/LightGBM
# 对于 windows 用户，需要执行：
# git submodule init
# git submodule update
cd LightGBM
mkdir build
cd build
cmake ..
make -j4
```

如果需要支持 `GPU`，则执行 `cmake` 时指定：

```
cmake -DUSE_GPU=1 ..
# 如果你已经安装了 NVIDIA OpenGL，则使用：
# cmake -DUSE_GPU=1 -DOpenCL_LIBRARY=/usr/local/cuda/lib64/libOpenCL.so -
DOpenCL_INCLUDE_DIR=/usr/local/cuda/include/ ..
```

如果想要 `protobuf` 来保存和加载模型，则先安装 `protobuf c++` 版本，然后使用：

```
cmake -DUSE_PROTO=ON ..
```

当前版本不支持该功能

- 安装 `python` 支持
  - 其中 `--precompile` 指示：已经编译过，不需要再次编译

```
cd python-package
sudo python setup.py install --precompile
```

### 2. 直接 `pip` 安装：

```

pip install lightgbm
pip install --no-binary :all: lightgbm #从源码编译安装
pip install lightgbm --install-option=--mpi #从源码编译安装 MPI 版本
pip install lightgbm --install-option=--gpu #从源码编译安装 GPU 版本
pip install lightgbm --install-option=--gpu --install-option="--opencl-include-dir=/usr/local/cuda/include/" --install-option="--opencl-library=/usr/local/cuda/lib64/libOpenCL.so" #从源码编译安装，指定配置
#可选的配置有：
#   boost-root
#   boost-dir
#   boost-include-dir
#   boost-librarydir
#   opencl-include-dir
#   opencl-library

```

## 二、调参

### 2.1 调参指导

#### 1. 针对 `leaf-wise` 树的参数优化：

- `num_leaves`：控制了叶节点的数目。它是控制树模型复杂度的主要参数。

如果是 `level-wise`，则该参数为  $2^{depth}$ ，其中  $depth$  为树的深度。

但是当叶子数量相同时，`leaf-wise` 的树要远远深过 `level-wise` 树，非常容易导致过拟合。因此应该让 `num_leaves` 小于  $2^{depth}$

在 `leaf-wise` 树中，并不存在 `depth` 的概念。因为不存在一个从 `leaves` 到 `depth` 的合理映射

- `min_data_in_leaf`：每个叶节点的最少样本数量。它是处理 `leaf-wise` 树的过拟合的重要参数。

将它设为较大的值，可以避免生成一个过深的树。但是也可能导致欠拟合。

- `max_depth`：控制了树的最大深度。

该参数可以显式的限制树的深度。

#### 2. 针对更快的训练速度：

- 通过设置 `bagging_fraction` 和 `bagging_freq` 参数来使用 bagging 方法
- 通过设置 `feature_fraction` 参数来使用特征的子抽样
- 使用较小的 `max_bin`
- 使用 `save_binary` 在未来的学习过程对数据加载进行加速

#### 3. 获取更好的准确率：

- 使用较大的 `max_bin`（学习速度可能变慢）
- 使用较小的 `learning_rate` 和较大的 `num_iterations`
- 使用较大的 `num_leaves`（可能导致过拟合）
- 使用更大的训练数据
- 尝试 `dart`

#### 4. 缓解过拟合：

- 使用较小的 `max_bin`
- 使用较小的 `num_leaves`
- 使用 `min_data_in_leaf` 和 `min_sum_hessian_in_leaf`
- 通过设置 `bagging_fraction` 和 `bagging_freq` 来使用 `bagging`
- 通过设置 `feature_fraction` 来使用特征子抽样
- 使用更大的训练数据
- 使用 `lambda_l1`, `lambda_l2` 和 `min_gain_to_split` 来使用正则
- 尝试 `max_depth` 来避免生成过深的树

## 2.2 参数

1. 参数的格式为 `key1=value1 key2=value2...`。（注意：这里是针对 `lightgbm` 可执行程序配置的。在 `PythonAPI` 中，可以直接设定）
  - 在配置文件和命令行中都可以设置参数。
  - 如果在命令行中设置参数时，在 `=` 前后都不应该有空格
  - 在使用配置文件设置参数时，一行只能包含一个参数。你可以使用 `#` 来注释。
  - 如果一个参数在命令行和配置文件中均出现，则优先使用命令行中的参数。

### 2.2.1 核心参数

1. `config` 或者 `config_file`：一个字符串，给出了配置文件的路径。默认为空字符串。
2. `task`：一个字符串，给出了要执行的任务。可以为：
  - `'train'` 或者 `'training'`：表示是训练任务。默认为 `'train'`。
  - `'predict'` 或者 `'prediction'` 或者 `'test'`：表示是预测任务。
  - `'convert_model'`：表示是模型转换任务。将模型文件转换成 `if-else` 格式。
3. `application` 或者 `objective` 或者 `app`：一个字符串，表示问题类型。可以为：
  - `'regression'` 或者 `'regression_l2'` 或者 `'mean_squared_error'` 或者 `'mse'` 或者 `'l2_root'` 或者 `'root_mean_squared_error'` 或者 `'rmse'`：表示回归任务，但是使用 `L2` 损失函数。默认为 `'regression'`
  - `'regression_l1'` 或者 `mae` 或者 `mean_absolute_error`：表示回归任务，但是使用 `L1` 损失函数。
  - `'huber'`：表示回归任务，但是使用 `huber` 损失函数。
  - `'fair'`：表示回归任务，但是使用 `fair` 损失函数。
  - `'poisson'`：表示 `Poisson` 回归任务。
  - `'quantile'`：表示 `quantile` 回归任务。
  - `'quantile_l2'`：表示 `quantile` 回归任务，但是使用了 `L2` 损失函数。
  - `'mape'` 或者 `'mean_absolute_percentage_error'`：表示回归任务，但是使用 `MAPE` 损失函数
  - `'gamma'`：表示 `gamma` 回归任务。
  - `'tweedie'`：表示 `tweedie` 回归任务。
  - `'binary'`：表示二分类任务，使用对数损失函数作为目标函数。
  - `'multiclass'`：表示多分类任务，使用 `softmax` 函数作为目标函数。必须设置 `num_class` 参数

- `'multiclassova'` 或者 `'multiclass_ova'` 或者 `'ova'` 或者 `'ovr'` : 表示多分类任务, 使用 `one-vs-all` 的二分类目标函数。必须设置 `num_class` 参数
- `'xentropy'` 或者 `'cross_entropy'` : 目标函数为交叉熵 (同时具有可选择的线性权重)。要求标签是  $[0,1]$  之间的数值。
- `'xentlambda'` 或者 `'cross_entropy_lambda'` : 替代了参数化的 `cross_entropy`。要求标签是  $[0,1]$  之间的数值。
- `'lambdarank'` : 表示排序任务。  
在 `lambdarank` 任务中, 标签应该为整数类型, 数值越大表示相关性越高。 `label_gain` 参数可以用于设置整数标签的增益 (权重)

4. `boosting` 或者 `'boost'` 或者 `'boosting_type'` : 一个字符串, 给出了基学习器模型算法。可以为:

- `'gbdt'` : 表示传统的梯度提升决策树。默认值为 `'gbdt'`
- `'rf'` : 表示随机森林。
- `'dart'` : 表示带 `dropout` 的 `gbdt`
- `goss` : 表示 Gradient-based One-Side Sampling 的 `gbdt`

5. `data` 或者 `train` 或者 `train_data` : 一个字符串, 给出了训练数据所在的文件的文件名。默认为空字符串。

`lightgbm` 将使用它来训练模型。

6. `valid` 或者 `test` 或者 `valid_data` 或者 `test_data` : 一个字符串, 表示验证集所在的文件的文件名。默认为空字符串。

`lightgbm` 将输出该数据集的度量。

如果有多个验证集, 则用逗号 , 分隔。

7. `num_iterations` 或者 `num_iteration` 或者 `num_tree` 或者 `num_trees` 或者 `num_round` 或者 `num_rounds` 或者 `num_boost_round` : 一个整数, 给出了 `boosting` 的迭代次数。默认为 100。

- 对于 `python/R` 包, 该参数是被忽略的。对于 `python`, 使用 `train()/cv()` 的输入参数 `num_boost_round` 来代替。
- 在内部, `lightgbm` 对于 `multiclass` 问题设置了 `num_class*num_iterations` 棵树。

8. `learning_rate` 或者 `shrinkage_rate` : 一个浮点数, 给出了学习率。默认为 0.1

在 `dart` 中, 它还会影响 `dropped trees` 的归一化权重。

9. `num_leaves` 或者 `num_leaf` : 一个整数, 给出了一棵树上的叶子数。默认为 31

10. `tree_learner` 或者 `tree` : 一个字符串, 给出了 `tree learner`, 主要用于并行学习。默认为 `'serial'`。可以为:

- `'serial'` : 单台机器的 `tree learner`
- `'feature'` : 特征并行的 `tree learner`
- `'data'` : 数据并行的 `tree learner`
- `'voting'` : 投票并行的 `tree learner`

11. `num_threads` 或者 `num_thread` 或者 `nthread` : 一个整数, 给出了 `lightgbm` 的线程数。默认为 `OpenMP_default`。

- 为了更快的速度, 应该将它设置为真正的 `CPU` 内核数, 而不是线程的数量 (大多数 `CPU` 使用超线程来使每个 `CPU` 内核生成2个线程)。

- 当数据集较小的时候，不要将它设置的过大
  - 对于并行学习，不应该使用全部的 CPU 核心，因为这会使得网络性能不佳
12. `device`：一个字符串，指定计算设备。默认为 `'cpu'`。可以为 `'gpu', 'cpu'`。
- 建议使用较小的 `max_bin` 来获得更快的计算速度
  - 为了加快学习速度，GPU 默认使用32位浮点数来求和。你可以设置 `gpu_use_dp=True` 来启动64位浮点数，但是它会使得训练速度降低。

## 2.2.2 学习控制参数

1. `max_depth`：一个整数，限制了树模型的最大深度，默认值为 `-1`。
  - 如果小于0，则表示没有限制。
2. `min_data_in_leaf` 或者 `min_data_per_leaf` 或者 `min_data` 或者 `min_child_samples`：一个整数，表示一个叶子节点上包含的最少样本数量。默认值为 20
3. `min_sum_hessian_in_leaf` 或者 `min_sum_hessian_per_leaf` 或者 `min_sum_hessian` 或者 `min_hessian` 或者 `min_child_weight`：一个浮点数，表示一个叶子节点上的最小 `hessian` 之和。（也就是叶节点样本权重之和的最小值）默认为 `1e-3`。
4. `feature_fraction` 或者 `sub_feature` 或者 `colsample_bytree`：一个浮点数，取值范围为 `[0.0, 1.0]`，默认值为 `1.0`。  
如果小于 `1.0`，则 `lightgbm` 会在每次迭代中随机选择部分特征。如 `0.8` 表示：在每棵树训练之前选择 80% 的特征来训练。
5. `feature_fraction_seed`：一个整数，表示 `feature_fraction` 的随机数种子，默认为2。
6. `bagging_fraction` 或者 `sub_row` 或者 `subsample`：一个浮点数，取值范围为 `[0.0, 1.0]`，默认值为 `1.0`。  
如果小于 `1.0`，则 `lightgbm` 会在每次迭代中随机选择部分样本来训练（非重复采样）。如 `0.8` 表示：在每棵树训练之前选择 80% 的样本（非重复采样）来训练。
7. `bagging_freq` 或者 `subsample_freq`：一个整数，表示每 `bagging_freq` 次执行 `bagging`。  
如果该参数为 0，表示禁用 `bagging`。
8. `bagging_seed` 或者 `bagging_fraction_seed`：一个整数，表示 `bagging` 的随机数种子，默认为 3。
9. `early_stopping_round` 或者 `early_stopping_rounds` 或者 `early_stopping`：一个整数，默认为0。  
如果一个验证集的度量在 `early_stopping_round` 循环中没有提升，则停止训练。如果为0则表示不开启早停。
10. `lambda_l1` 或者 `reg_alpha`：一个浮点数，表示 L1 正则化系数。默认为0
11. `lambda_l2` 或者 `reg_lambda`：一个浮点数，表示 L2 正则化系数。默认为0
12. `min_split_gain` 或者 `min_gain_to_split`：一个浮点数，表示执行切分的最小增益，默认为0
13. `drop_rate`：一个浮点数，取值范围为 `[0.0, 1.0]`，表示 `dropout` 的比例，默认为0.1。该参数仅在 `dart` 中使用
14. `skip_drop`：一个浮点数，取值范围为 `[0.0, 1.0]`，表示跳过 `dropout` 的概率，默认为0.5。该参数仅在 `dart` 中使用

15. `max_drop` : 一个整数, 表示一次迭代中删除树的最大数量, 默认为50。如果小于等于0, 则表示没有限制。该参数仅在 `dart` 中使用
16. `uniform_drop` : 一个布尔值, 表示是否想要均匀的删除树, 默认值为 `False`。该参数仅在 `dart` 中使用
17. `xgboost_dart_mode` : 一个布尔值, 表示是否使用 `xgboost dart` 模式, 默认值为 `False`。该参数仅在 `dart` 中使用
18. `drop_seed` : 一个整数, 表示 `dropout` 的随机数种子, 默认值为 4。该参数仅在 `dart` 中使用
19. `top_rate` : 一个浮点数, 取值范围为 `[0.0,1.0]`, 表示在 `goss` 中, 大梯度数据的保留比例, 默认值为 0.2。该参数仅在 `goss` 中使用
20. `other_rate` : 一个浮点数, 取值范围为 `[0.0,1.0]`, 表示在 `goss` 中, 小梯度数据的保留比例, 默认值为 0.1。该参数仅在 `goss` 中使用
21. `min_data_per_group` : 一个整数, 表示每个分类组的最小数据量, 默认值为100。用于排序任务
22. `max_cat_threshold` : 一个整数, 表示 `category` 特征的取值集合的最大大小。默认为 32。
23. `cat_smooth` : 一个浮点数, 用于 `category` 特征的概率平滑。默认值为 10。  
它可以降低噪声在 `category` 特征中的影响, 尤其是对于数据很少的类。
24. `cat_l2` : 一个浮点数, 用于 `category` 切分中的 L2 正则化系数。默认为 10。
25. `top_k` 或者 `topk` : 一个整数, 用于投票并行中。默认为20。  
将它设置为更大的值可以获得更精确的结果, 但是会降低训练速度。

### 2.2.3 IO 参数

1. `max_bin` : 一个整数, 表示最大的桶的数量。默认值为 255。
  - `lightgbm` 会根据它来自动压缩内存。如 `max_bin=255` 时, 则 `lightgbm` 将使用 `uint8` 来表示特征的每一个值。
2. `min_data_in_bin` : 一个整数, 表示每个桶的最小样本数。默认为3。  
该方法可以避免出现一个桶只有一个样本的情况。
3. `data_random_seed` : 一个整数, 表示并行学习数据分隔中的随机数种子。默认为1  
它不包括特征并行。
4. `output_model` 或者 `model_output` 或者 `model_out` : 一个字符串, 表示训练中输出的模型被保存的文件的文件名。默认 `LightGBM_model.txt`。
5. `input_model` 或者 `model_input` 或者 `model_in` : 一个字符串, 表示输入模型的文件的文件名。默认空字符串。
  - 对于 `prediction` 任务, 该模型将用于预测数据
  - 对于 `train` 任务, 训练将从该模型继续
6. `output_result` 或者 `predict_result` 或者 `prediction_result` : 一个字符串, 给出了 `prediction` 结果存放的文件名。默认为 `LightGBM_predict_result.txt`。
7. `pre_partition` 或者 `is_pre_partition` : 一个布尔值, 指示数据是否已经被划分。默认值为 `False`。如果为 `true`, 则不同的机器使用不同的 `partition` 来训练。  
它用于并行学习 (不包括特征并行)

8. `is_sparse` 或者 `is_enable_sparse` 或者 `enable_sparse` : 一个布尔值, 表示是否开启稀疏优化, 默认为 `True`。

如果为 `True` 则启用稀疏优化。

9. `two_round` 或者 `two_round_loading` 或者 `use_two_round_loading` : 一个布尔值, 指示是否启动两次加载。默认值为 `False`, 表示只需要进行一次加载。

- 默认情况下, `lightgbm` 会将数据文件映射到内存, 然后从内存加载特征, 这将提供更快的数据加载速度。但是当数据文件很大时, 内存可能会被耗尽。
- 如果数据文件太大, 则将它设置为 `True`

10. `save_binary` 或者 `is_save_binary` 或者 `is_save_binary_file` : 一个布尔值, 表示是否将数据集 (包括验证集) 保存到二进制文件中。默认值为 `False`。

如果为 `True`, 则可以加快数据的加载速度。

11. `verbosity` 或者 `verbose` : 一个整数, 表示是否输出中间信息。默认值为1。

- 如果小于0, 则仅仅输出 `critical` 信息; 如果等于0, 则还会输出 `error,warning` 信息; 如果大于0, 则还会输出 `info` 信息。

12. `header` 或者 `has_header` : 一个布尔值, 表示输入数据是否有头部。默认为 `False`。

13. `label` 或者 `label_column` : 一个字符串, 表示标签列。默认为空字符串。

- 你也可以指定一个整数, 如 `label=0` 表示第0列是标签列。
- 你也可以为列名添加前缀, 如 `label=prefix:label_name`

14. `weight` 或者 `weight_column` : 一个字符串, 表示样本权重列。默认为空字符串。

- 你也可以指定一个整数, 如 `weight=0` 表示第0列是权重列。注意: 它是剔除了标签列之后的索引。假如标签列为 `0`, 权重列为 `1`, 则这里 `weight=0`。
- 你也可以为列名添加前缀, 如 `weight=prefix:weight_name`

15. `query` 或者 `query_column` 或者 `group` 或者 `group_column` : 一个字符串, `query/group ID` 列。默认为空字符串。

- 你也可以指定一个整数, 如 `query=0` 表示第0列是 `query` 列。注意: 它是剔除了标签列之后的索引。假如标签列为 `0`, `query` 列为 `1`, 则这里 `query=0`。
- 你也可以为列名添加前缀, 如 `query=prefix:query_name`

16. `ignore_column` 或者 `ignore_feature` 或者 `blacklist` : 一个字符串, 表示训练中忽略的一些列, 默认为空字符串。

- 可以用数字做索引, 如 `ignore_column=0,1,2` 表示第0,1,2 列将被忽略。注意: 它是剔除了标签列之后的索引。
- 你也可以为列名添加前缀, 如 `ignore_column=prefix:ign_name1,ign_name2`

17. `categorical_feature` 或者 `categorical_column` 或者 `cat_feature` 或者 `cat_column` : 一个字符串, 指定 `category` 特征的列。默认为空字符串。

- 可以用数字做索引, 如 `categorical_feature=0,1,2` 表示第0,1,2 列将作为 `category` 特征。注意: 它是剔除了标签列之后的索引。
- 你也可以为列名添加前缀, 如 `categorical_feature=prefix:cat_name1,cat_name2`
- 在 `categorical` 特征中, 负的取值被视作缺失值。



18. `predict_raw_score` 或者 `raw_score` 或者 `is_predict_raw_score` : 一个布尔值, 表示是否预测原始得分。默认为 `False`。
- 如果为 `True` 则仅预测原始得分。
- 该参数只用于 `prediction` 任务。
19. `predict_leaf_index` 或者 `leaf_index` 或者 `is_predict_leaf_index` : 一个布尔值, 表示是否预测每个样本在每棵树上的叶节点编号。默认为 `False`。
- 在预测时, 每个样本都会被分配到每棵树的某个叶子节点上。该参数就是要输出这些叶子节点的编号。
- 该参数只用于 `prediction` 任务。
20. `predict_contrib` 或者 `contrib` 或者 `is_predict_contrib` : 一个布尔值, 表示是否输出每个特征对于每个样本的预测的贡献。默认为 `False`。
- 输出的结果形状为 `[nsamples,nfeatures+1]`, 之所以 `+1` 是考虑到 `bias` 的贡献。所有的贡献加起来就是该样本的预测结果。
- 该参数只用于 `prediction` 任务。
21. `bin_construct_sample_cnt` 或者 `subsample_for_bin` : 一个整数, 表示用来构建直方图的样本的数量。默认为 `200000`。
- 如果数据非常稀疏, 则可以设置为一个更大的值
  - 如果设置更大的值, 则会提供更好的训练效果, 但是会增加数据加载时间
22. `num_iteration_predict` : 一个整数, 表示在预测中使用多少棵树。默认为 `-1`。
- 小于等于 `0` 表示使用模型的所有子树。
- 该参数只用于 `prediction` 任务。
23. `pred_early_stop` : 一个布尔值, 表示是否使用早停来加速预测。默认为 `False`。
- 如果为 `True`, 则可能影响精度。
24. `pred_early_stop_freq` : 一个整数, 表示检查早停的频率。默认为 `10`
25. `pred_early_stop_margin` : 一个浮点数, 表示早停的边际阈值。默认为 `10.0`
26. `use_missing` : 一个布尔值, 表示是否使用缺失值功能。默认为 `True`
- 如果为 `False` 则禁用缺失值功能。
27. `zero_as_missing` : 一个布尔值, 表示是否将所有的零 (包括在 `libsvm/sparse` 矩阵 中未显示的值) 都视为缺失值。默认为 `False`。
- 如果为 `False`, 则将 `np.nan` 视作缺失值。
  - 如果为 `True`, 则 `np.nan` 和 零都将视作缺失值。
28. `init_score_file` : 一个字符串, 表示训练时的初始化分数文件的路径。默认为空字符串, 表示 `train_data_file+".init"` (如果存在)
29. `valid_init_score_file` : 一个字符串, 表示验证时的初始化分数文件的路径。默认为空字符串, 表示 `valid_data_file+".init"` (如果存在)
- 如果有多个 (对应于多个验证集), 则可以用逗号, 来分隔。

## 2.2.4 目标函数的参数



1. `sigmoid` : 一个浮点数, 用 `sigmoid` 函数的参数, 默认为 1.0。  
它用于二分类任务和 `lambdarank` 任务。
2. `alpha` : 一个浮点数, 用于 `Huber` 损失函数和 `Quantile regression` , 默认值为 1.0。  
它用于 `huber` 回归任务和 `Quantile` 回归任务。
3. `fair_c` : 一个浮点数, 用于 `Fair` 损失函数, 默认值为 1.0。  
它用于 `fair` 回归任务。
4. `gaussian_eta` : 一个浮点数, 用于控制高斯函数的宽度, 默认值为 1.0。  
它用于 `regression_l1` 回归任务和 `huber` 回归任务。
5. `poisson_max_delta_step` : 一个浮点数, 用于 `Poisson regression` 的参数, 默认值为 0.7。  
它用于 `poisson` 回归任务。
6. `scale_pos_weight` : 一个浮点数, 用于调整正样本的权重, 默认值为 1.0  
它用于二分类任务。
7. `boost_from_average` : 一个布尔值, 指示是否将初始得分调整为平均值 ( 它可以使得收敛速度更快 )。  
它。默认为 `True`。  
它用于回归任务。
8. `is_unbalance` 或者 `unbalanced_set` : 一个布尔值, 指示训练数据是否均衡的。默认为 `True`。  
它用于二分类任务。
9. `max_position` : 一个整数, 指示将在这个 `NDCG` 位置优化。默认为 20。  
它用于 `lambdarank` 任务。
10. `label_gain` : 一个浮点数序列, 给出了每个标签的增益。默认值为 `0,1,3,7,15,...` ( 即  $a_n = 2^{n-1} - 1$  )  
它用于 `lambdarank` 任务。
11. `num_class` 或者 `num_classes` : 一个整数, 指示了多分类任务中的类别数量。默认为 1  
它用于多分类任务。
12. `reg_sqrt` : 一个布尔值, 默认为 `False`。  
如果为 `True` , 则拟合的结果为:  $\sqrt{\text{label}}$ 。同时预测的结果被自动转换为:  $\text{pred}^2$ 。  
它用于回归任务。

## 2.2.5 度量参数

1. `metric` : 一个字符串, 指定了度量的指标, 默认为: 对于回归问题, 使用 `l2` ; 对于二分类问题, 使用 `binary_logloss` ; 对于 `lambdarank` 问题, 使用 `ndcg` 。
  - o `'l1'` 或者 `mean_absolute_error` 或者 `mae` 或者 `regression_l1` : 表示绝对值损失
  - o `'l2'` 或者 `mean_squared_error` 或者 `mse` 或者 `regression_l2` 或者 `regression` : 表示平方损失
  - o `'l2_root'` 或者 `root_mean_squared_error` 或者 `rmse` : 表示开方损失
  - o `'quantile'` : 表示 `Quantile` 回归中的损失
  - o `'mape'` 或者 `'mean_absolute_percentage_error'` : 表示 `MAPE` 损失

- 'huber' : 表示 huber 损失
- 'fair' : 表示 fair 损失
- 'poisson' : 表示 poisson 回归的负对数似然
- 'gamma' : 表示 gamma 回归的负对数似然
- 'gamma\_deviance' : 表示 gamma 回归的残差的方差
- 'tweedie' : 表示 Tweedie 回归的负对数似然
- 'ndcg' : 表示 NDCG
- 'map' 或者 'mean\_average\_precision' : 表示平均的精度
- 'auc' : 表示 AUC
- 'binary\_logloss' 或者 'binary' : 表示二类分类中的对数损失函数
- 'binary\_error' : 表示二类分类中的分类错误率
- 'multi\_logloss' 或者 'multiclass' 或者 'softmax' 或者 'multiclassova' 或者 'multiclass\_ova' 或者 'ova' 或者 'ovr' : 表示多类分类中的对数损失函数
- 'multi\_error' : 表示多分类中的分类错误率
- 'xentropy' 或者 'cross\_entropy' : 表示交叉熵
- 'xentlambd' 或者 'cross\_entropy\_lambda' : 表示 intensity 加权的交叉熵
- 'kldiv' 或者 'kullback\_leibler' : 表示 KL 散度

如果有多个度量指标，则用逗号，分隔。

2. metric\_freq 或者 'output\_freq' : 一个正式，表示每隔多少次输出一度度量结果。默认为1。
3. train\_metric 或者 training\_metric 或者 is\_training\_metric : 一个布尔值，默认为 False。

如果为 True，则在训练时就输出度量结果。

4. ndcg\_at 或者 ndcg\_eval\_at 或者 eval\_at : 一个整数列表，指定了 NDCG 评估点的位置。默认为 1,2,3,4,5。

## 2.2.6 网络参数

这里的参数仅用于并行学习，并且仅用于 socket 版本，而不支持 mpi 版本。

1. num\_machines 或者 num\_machine : 一个整数，表示并行学习的机器的数量。默认为 1。
  2. local\_listen\_port 或者 local\_port : 一个整数，表示监听本地机器的端口号。默认为 12400。
- 训练之前，你应该在防火墙中开放该端口。
3. time\_out : 一个整数，表示允许 socket 的超时时间（单位：分钟）。默认值为 120
  4. machine\_list\_file 或者 mlist : 一个字符串，指示了并行学习的机器列表文件的文件名。默认为空字符串。

该文件每一行包含一个 IP 和端口号。格式是 ip port，以空格分隔。

## 2.2.7 GPU 参数

1. gpu\_platform\_id : 一个整数，表示 OpenCL platform ID。通常每个 GPU 供应商都会公开一个 OpenCL platform。默认为-1（表示系统级的默认 platform）
2. gpu\_device\_id : 一个整数，表示设备 ID。默认为-1。

在选定的 platform 上，每个 GPU 都有一个唯一的设备 ID。-1 表示选定 platform 上的默认设备。

3. `gpu_use_dp`：一个布尔值，默认值为 `False`。
- 如果为 `True`，则在 `GPU` 上计算时使用双精度（否则使用单精度）

2.2.8 模型参数

1. `convert_model_language`：一个字符串，目前只支持 `'cpp'`。
- 如果该参数被设置，且 `task='train'`，则该模型会被转换。
2. `convert_model`：一个字符串，表示转换模型到一个文件的文件名。默认为 `'gbdt_prediction.cpp'`

三、进阶

3.1 缺失值处理

1. `lightgbm` 默认处理缺失值，你可以通过设置 `use_missing=False` 使其无效。
2. `lightgbm` 默认使用 `NaN` 来表示缺失值。你可以设置 `zero_as_missing` 参数来改变其行为：
- `zero_as_missing=True` 时：

`NaN` 和 `0`（包括在稀疏矩阵里，没有显示的值）都视作缺失值。

`zero_as_missing=False` 时：

只有 `NaN` 才是缺失值（默认的行为）

3.2 分类特征支持

1. 当使用 `local categorical` 特征（而不是 `one-hot` 编码的特征）时，`lightgbm` 可以提供良好的精确度。
2. 要想使用 `categorical` 特征，则启用 `categorical_feature` 参数（参数值为列名字符串或者列名字符串的列表）
- 首先要将 `categorical` 特征的取值转换为非负整数，而且如果是连续的范围更好

然后使用 `min_data_per_group` 和 `cat_smooth` 去处理过拟合（当样本数较小，或者 `category` 取值范围较大时）

3.3 LambdaRank

1. `LambdaRank` 中，标签应该是整数类型，较大的数字代表更高的相关性。
- 如：`0` 表示差、`1` 表示一般、`2` 表示好、`3` 表示完美
2. 使用 `max_position` 设置 `NDCG` 优化的位置
3. `label_gain` 设置增益的 `int` 标签。

3.4 并行学习

1. `lightgbm` 已经提供了以下并行学习算法：

并行算法	开启方式
数据并行	<code>tree_learner='data'</code>
特征并行	<code>tree_learner='feature'</code>
投票并行	<code>tree_learner='voting'</code>

`tree_learner` 默认为 `'serial'`。表示串行学习。

这些并行算法适用于不同场景：

	样本数量较小	样本数量巨大
特征数量较小	特征并行	数据并行
特征数量巨大	特征并行	投票并行

2. 构建并行版本：

默认的并行版本基于 `socket` 的并行学习，如果需要基于 `MPI` 的并行版本，则需要手动编译

- 首先收集所有想要运行并行学习的机器的 `IP`，并指定一个 `TCP` 端口号（要求在这些机器上，这些端口没有被防火墙屏蔽掉）。

然后将这些 `IP` 和端口写入到文件中（假设文件名为 `ip.txt`）：

```
ip1 port
ip2 port
```

其中要求：

- 数量必须和 `num_machines` 或者 `num_machine` 参数相等
  - 必须包含 `127.0.0.1`（或者包含 `localhost` 对应的其它 `ip`），它代表本地
  - `port` 必须和 `local_listen_port` 或者 `local_port` 参数相等
- 然后在配置文件中编译以下参数：

```
tree_learner= 你的并行算法
num_machines= 并行计算的机器的数量
machine_list_file=ip.txt #要求每个并行计算的机器占一行
local_listen_port=port
```

- 然后将数据文件、可执行文件、配置文件、以及 `ip.txt` 拷贝到所有并行学习的机器上。
- 在所有机器上运行以下命令：
  - `windows`： `lightgbm.exe config=配置文件`
  - `Linux`： `./lightgbm config=配置文件`

3. 并行学习的注意事项：

- 当前 `Python` 版本不支持并行，必须采用 `lightgbm` 二进制的方式。
- 在执行推断时，要求数据的特征与训练时的特征完全一致
  - 必须都包含 `label` 列。推断时，该列的数值不起作用，仅仅是个占位符。
  - 如果有 `header`，则列的先后顺序不重要。如果没有 `header`，则必须顺序也要保持相同。

# 四、API

## 4.1 数据接口

### 4.1.1 数据格式

1. `lightgbm` 支持 `csv`, `tsv`, `libsvm` 格式的输入数据文件。其中：

- 可以包含标题
- 可以指定 `label` 列、权重列、`query/group id` 列
- 也可以指定一个被忽略的列的列表。

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

2. `lightgbm` 也支持 `numpy 2d array` 以及 `pandas` 对象。

```
data = np.random.rand(500, 10) # 500 个样本，每一个包含 10 个特征
label = np.random.randint(2, size=500) # 二元目标变量，0 和 1
train_data = lgb.Dataset(data, label=label)
```

3. `lightgbm` 也支持 `scipy.sparse.csr_matrix`：

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

4. `lightgbm` 可以通过 `Lightgbm` 二进制文件来保存和加载数据。

```
train_data = lgb.Dataset('train.bin')
train_data.save_binary('train2.bin')
```

5. 创建验证集：（要求验证集和训练集的格式一致）

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

或者：

```
train_data = lgb.Dataset('train.svm.txt')
test_data = train_data.create_valid('test.svm')
```

6. `lightgbm` 中的 `Dataset` 对象由于仅仅需要保存离散的数据桶，因此它具有很好的内存效率。

但是由于 `numpy array/pandas` 对象的内存开销较大，因此当使用它们来创建 `Dataset` 时，你可以通过下面的方式来节省内存：

- 构造 `Dataset` 时，设置 `free_raw_data=True`
- 在构造 `Dataset` 之后，手动设置 `raw_data=True`
- 手动调用 `gc`

7. `categorical_feature` 的支持：

- 需要指定 `categorical_feature` 参数

- 对于 `categorical_feature` 特征，首选需要将它转换为整数类型，并且只支持非负数。如果转换为连续的范围更佳。

## 4.1.2 Dataset

- `Dataset`：由 `lightgbm` 内部使用的数据结构，它存储了数据集。

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

- 参数：

- `data`：一个字符串、`numpy array` 或者 `scipy.parse`，它指定了数据源。  
如果是字符串，则表示数据源文件的文件名。
- `label`：一个列表、1维的 `numpy array` 或者 `None`，它指定了样本标记。默认为 `None`。
- `max_bin`：一个整数或者 `None`，指定每个特征的最大分桶数量。默认为 `None`。  
如果为 `None`，则从配置文件中读取。
- `reference`：一个 `Dataset` 或者 `None`。默认为 `None`。  
如果当前构建的数据集用于验证集，则 `reference` 必须传入训练集。否则会报告 `has different bin mappers`。
- `weight`：一个列表、1维的 `numpy array` 或者 `None`，它指定了样本的权重。默认为 `None`。
- `group`：一个列表、1维的 `numpy array` 或者 `None`，它指定了数据集的 `group/query size`。默认为 `None`。
- `init_score`：一个列表、1维的 `numpy array` 或者 `None`，它指定了 `Booster` 的初始 `score`。默认为 `None`。
- `silent`：一个布尔值，指示是否在构建过程中输出信息。默认为 `False`。
- `feature_name`：一个字符串列表或者 `'auto'`，它指定了特征的名字。默认为 `'auto'`。
  - 如果数据源为 `pandas DataFrame` 并且 `feature_name='auto'`，则使用 `DataFrame` 的 `column names`。
- `categorical_feature`：一个字符串列表、整数列表、或者 `'auto'`。它指定了 `categorical` 特征。默认为 `'auto'`。
  - 如果是整数列表，则给定了 `categorical` 特征的下标。
  - 如果是字符串列表，在给定了 `categorical` 特征的名字。此时必须设定 `feature_name` 参数。
  - 如果是 `'auto'` 并且数据源为 `pandas DataFrame`，则 `DataFrame` 的 `categorical` 列将作为 `categorical` 特征。
- `params`：一个字典或者 `None`，指定了其它的参数。默认为 `None`。
- `free_raw_data`：一个布尔值，指定是否在创建完 `Dataset` 之后释放原始的数据。默认为 `True`。  
调用 `Dataset()` 之后，并没有构建完 `Dataset`。构建完需要等到构造一个 `Booster` 的时候。

- 方法：



- `.construct()` : 延迟初始化函数。它返回当前的 `Dataset` 本身
- `.create_valid(data, label=None, weight=None, group=None, init_score=None, silent=False, params=None)` : 创建一个验证集 ( 其格式与当前的 `Dataset` 相同 )
  - 参数: 参考 `Dataset` 的初始化函数
  - 返回值: 当前的 `Dataset` 本身
- `.get_field(field_name)` : 获取当前 `Dataset` 的属性
 

它要求 `Dataset` 已经构建完毕。否则抛出 `Cannot get group before construct Dataset` 异常。

  - 参数: `field_name` : 一个字符串, 指示了属性的名字
  - 返回值: 一个 `numpy array`, 表示属性的值。如果属性不存在则返回 `None`
- `.set_field(field_name, data)` : 设置当前 `Dataset` 的属性
  - 参数:
    - `field_name` : 一个字符串, 指示了属性的名字
    - `data` : 一个列表、`numpy array` 或者 `None`, 表示属性的值
- `.get_group()` : 获取当前 `Dataset` 的 `group`

`get_xxx()` 等方法, 都是调用的 `get_field()` 方法来实现的

  - 返回值: 一个 `numpy array`, 表示每个分组的 `size` 。
- `.set_group(group)` : 设置当前 `Dataset` 的 `group`
  - 参数: `group` : 一个列表、`numpy array` 或者 `None`, 表示每个分组的 `size` 。
- `.get_init_score()` : 获取当前 `Dataset` 的初始化 `score`

`get_xxx()` 等方法, 都是调用的 `get_field()` 方法来实现的

  - 返回值: 一个 `numpy array`, 表示 `Booster` 的初始化 `score`
- `.set_init_score(init_score)` : 设置 `Booster` 的初始化 `score`
  - 参数: `init_score` : 一个列表、`numpy array` 或者 `None`, 表示 `Booster` 的初始化 `score`
- `.get_label()` : 获取当前 `Dataset` 的标签
 

`get_xxx()` 等方法, 都是调用的 `get_field()` 方法来实现的

  - 返回值: 一个 `numpy array`, 表示当前 `Dataset` 的标签信息
- `.set_label(label)` : 设置当前 `Dataset` 的标签
  - 参数: `label` : 一个列表、`numpy array` 或者 `None`, 表示当前 `Dataset` 的标签信息
- `.get_ref_chain(ref_limit=100)` : 获取 `Dataset` 对象的 `reference` 链。
 

假设 `d` 为一个 `Dataset` 对象, 则只要 `d.reference` 存在, 则获取 `d.reference` ; 只要 `d.reference.reference` 存在, 则获取 `d.reference.reference` ...

  - 参数: `ref_limit` : 一个整数, 表示链条的最大长度
  - 返回值: 一个 `Dataset` 的集合
- `.set_reference(reference)` : 设置当前 `Dataset` 的 `reference`
  - 参数: `reference` : 另一个 `Dataset` 对象, 它作为创建当前 `Dataset` 的模板
- `.get_weight()` : 返回 `Dataset` 中每个样本的权重
 

`get_xxx()` 等方法, 都是调用的 `get_field()` 方法来实现的

- 返回值：一个 `numpy array`，表示当前 `Dataset` 每个样本的权重
  - `.set_weight(weight)`：设置 `Dataset` 中每个样本的权重
    - 参数：`weight`：一个列表、`numpy array` 或者 `None`，表示当前 `Dataset` 每个样本的权重
  - `.num_data()`：返回 `Dataset` 中的样本数量
  - `.num_feature()`：返回 `Dataset` 中的特征数量
  - `.save_binary(filename)`：以二进制文件的方式保存 `Dataset`
    - 参数：`filename`：保存的二进制文件的文件名
  - `.set_categorical_feature(categorical_feature)`：设置 `categorical` 特征
    - 参数：`categorical_feature`：一个字符串列表或者整数列表。给出了 `categorical` 特征的名字，或者给出了 `categorical` 特征的下标
  - `.set_feature_name(feature_name)`：设置特征名字
    - 参数：`feature_name`：一个字符串列表。给出了特征名字
  - `.subset(used_indices, params=None)`：获取当前 `Dataset` 的一个子集
    - 参数：
      - `used_indices`：一个整数的列表，它给出了当前 `Dataset` 中样本的下标。这些样本将构建子集
      - `params`：一个字典或者 `None`，给出了其它的参数。默认为 `None`
    - 返回值：一个新的 `Dataset` 对象。
3. 当你通过 `Dataset()` 来创建一个 `Dataset` 对象的时候，它并没有真正的创建必要的数据（必要的数据指的是为训练、预测等准备好的数据），而是推迟到构造一个 `Booster` 的时候。

因为 `lightgbm` 需要构造 `bin mappers` 来建立子树、建立同一个 `Booster` 内的训练集和验证集（训练集和验证集共享同一个 `bin mappers`、`categorical features`、`feature names`）。所以 `Dataset` 真实的数据推迟到了构造 `Booster` 的时候。

在构建 `Dataset` 之前：

- `get_label()`、`get_weight()`、`get_init_score()`、`get_group()`：等效于 `self.label`、`self.weight`、`self.init_score`、`self.group`
- 此时调用 `self.get_field(field)` 会抛出异常：`Cannot get group before construct Dataset`
- `set_label()`、`set_weight()`、`set_init_score()`、`set_group()`：等效于 `self.label=xxx`、`self.weight=xxx`、`self.init_score=xxx`、`self.group=xxx`
  - `self.num_data()`、`self._num_feature()` 可以从 `self.data` 中获取。
- 如果 `self.data` 是 `ndarray`，则它们就是 `self.data.shape[0]`、`self.data.shape[1]`

4. 示例：

```
import lightgbm as lgb
import numpy as np
class DatasetTest:
    def __init__(self):
        self._matrix1 = lgb.Dataset('data/train.svm.txt')
```

```

self._matrix2 = lgb.Dataset(data=np.arange(0, 12).reshape((4, 3)),
                             label=[1, 2, 3, 4], weight=[0.5, 0.4, 0.3, 0.2],
                             silent=False, feature_name=['a', 'b', 'c'])

def print(self, matrix):
    """
    Matrix 构建尚未完成时的属性
    :param matrix:
    :return:
    """
    print('data: %s' % matrix.data)
    print('label: %s' % matrix.label)
    print('weight: %s' % matrix.weight)
    print('init_score: %s' % matrix.init_score)
    print('group: %s' % matrix.group)

def run_method(self, matrix):
    """
    测试一些 方法
    :param matrix:
    :return:
    """
    print('get_ref_chain():', matrix.get_ref_chain(ref_limit=10))
    # get_ref_chain(): {<lightgbm.basic.Dataset object at 0x7f29cd762f28>}
    print('subset():', matrix.subset(used_indices=[0, 1]))
    # subset(): <lightgbm.basic.Dataset object at 0x7f29a4aeb518>

def test(self):
    self.print(self._matrix1)
    # data: data/train.svm.txt
    # label: None
    # weight: None
    # init_score: None
    # group: None

    self.print(self._matrix2)
    # data: [[ 0  1  2]
    # [ 3  4  5]
    # [ 6  7  8]
    # [ 9 10 11]]
    # label: [1, 2, 3, 4]
    # weight: [0.5, 0.4, 0.3, 0.2]
    # init_score: No

    self.run_method(self._matrix2)

```

5. 你要确保你的数据集的样本数足够大，从而满足一些限制条件（如：单个节点的最小样本数、单个桶的最小样本数等）。否则会直接报错。

## 4.2 模型接口

## 4.2.1 Booster

### 1. LightGBM 中的 Booster 类：

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

参数：

- `params`：一个字典或者 `None`，给出了 `Booster` 的参数。默认为 `None`
- `train_set`：一个 `Dataset` 对象或者 `None`，给出了训练集。默认为 `None`
- `model_file`：一个字符串或者 `None`，给出了 `model file` 的路径。默认为 `None`
- `silent`：一个布尔值，指示是否在构建过程中打印消息。默认为 `False`

### 2. 方法：

- `.add_valid(data,name)`：添加一个验证集。
  - 参数：
    - `data`：一个 `Dataset`，代表一个验证集
    - `name`：一个字符串，表示该验证集的名字。不同的验证集通过名字来区分
- `.attr(key)`：获取属性的值。
  - 参数：`key`：一个字符串，表示属性的名字
  - 返回值：该属性的名字。如果属性不存在则返回 `None`
- `.current_iteration()`：返回当前的迭代的 `index`（即迭代的编号）
- `.dump_model(num_iteration=-1)`：dump 当前的 `Booster` 对象为 `json` 格式。
  - 参数：`num_iteration`：一个整数，指定需要 dump 第几轮训练的结果。如果小于0，则最佳迭代步的结果（如果存在的话）将被 dump。默认为 `-1`。
  - 返回值：一个字典，它表示 dump 之后的 `json`
- `.eval(data,name,feval=None)`：对指定的数据集 evaluate
  - 参数：
    - `data`：一个 `Dataset` 对象，代表被评估的数据集
    - `name`：一个字符串，表示被评估的数据集的名字。不同的验证集通过名字来区分
    - `feval`：一个可调用对象或者 `None`，它表示自定义的 `evaluation` 函数。默认为 `None`。它的输入为 `(y_true, y_pred)`、或者 `( y_true, y_pred, weight)`、或者 `(y_true, y_pred, weight, group)`，返回一个元组：`(eval_name,eval_result,is_higher_better)`。或者返回该元组的列表。
  - 返回值：一个列表，给出了 `evaluation` 的结果。
- `.eval_train(feval=None)`：对训练集进行 evaluate
  - 参数：`feval`：参考 `.eval()` 方法
  - 返回值：一个列表，给出了 `evaluation` 的结果。
- `.eval_valid(feval=None)`：对验证集进行 evaluate
  - 参数：`feval`：参考 `.eval()` 方法
  - 返回值：一个列表，给出了 `evaluation` 的结果。
- `.feature_importance(importance_type='split', iteration=-1)`：获取特征的 `importance`

- 参数：
  - `importance_type`：一个字符串，给出了特征的 `importance` 衡量指标。默认为 `'split'`。可以为：
    - `'split'`：此时特征重要性衡量标准为：该特征在所有的树中，被用于划分数据集的总次数。
    - `'gain'`：此时特征重要性衡量标准为：该特征在所有的树中获取的总收益。
  - `iteration`：一个整数，指定需要考虑的是第几轮训练的结果。如果小于0，则最佳迭代步的结果（如果存在的话）将被考虑。默认为 `-1`。
- 返回值：一个 `numpy array`，表示每个特征的重要性
- `.feature_name()`：获取每个特征的名字。
  - 返回值：一个字符串的列表，表示每个特征的名字
- `.free_dataset()`：释放 `Booster` 对象的数据集
- `.free_network()`：释放 `Booster` 对象的 `Network`
- `.get_leaf_output(tree_id, leaf_id)`：获取指定叶子的输出
  - 输入：
    - `tree_id`：一个整数，表示子学习器的编号
    - `leaf_id`：一个整数，表示该子学习器的叶子的编号
  - 返回值：一个浮点数，表示该叶子节点的输出
- `.num_feature()`：获取特征的数量（即由多少列特征）
- `.predict(data, num_iteration=-1, raw_score=False, pred_leaf=False, pred_contrib=False, data_has_header=False, is_reshape=True, pred_parameter=None)`：执行预测
  - 参数：
    - `data`：一个字符串、`numpy array` 或者 `scipy.sparse`，表示被测试的数据集。如果为字符串，则表示测试集所在的文件的文件名。  
注意：如果是 `numpy array` 或者 `pandas dataframe` 时，要求数据的列必须与训练时的列顺序一致。
    - `num_iteration`：一个整数，表示用第几轮的迭代结果来预测。如果小于0，则最佳迭代步的结果（如果存在的话）将被使用。默认为 `-1`。
    - `raw_score`：一个布尔值，指示是否输出 `raw scores`。默认为 `False`
    - `pred_leaf`：一个布尔值。如果为 `True`，则会输出每个样本在每个子树的哪个叶子上。它是一个 `nsample x ntrees` 的矩阵。默认为 `False`。  
每个子树的叶节点都是从 1 开始编号的。
    - `pred_contrib`：一个布尔值。如果为 `True`，则输出每个特征对每个样本预测结果的贡献程度。它是一个 `nsample x (nfeature+1)` 的矩阵。默认为 `False`。  
之所以加1，是因为有 `bias` 的因素。它位于最后一列。  
其中样本所有的贡献程度相加，就是该样本最终的预测的结果。
    - `data_has_header`：一个布尔值，指示数据集是否含有标题。仅当 `data` 是字符串时有效。默认为 `False`。

- `is_reshape` : 一个布尔值, 指示是否 `reshape` 结果成 `[nrow, ncol]`。默认为 `True`
- `pred_parameter` : 一个字典或者 `None`, 给出了其它的参数。默认为 `None`
- 返回值: 一个 `numpy array`, 表示预测结果
- `.reset_parameter(params)` : 重设 `Booster` 的参数。
  - 参数: `params` : 一个字典, 给出了新的参数
- `.rollback_one_iter()` : 将 `Booster` 回滚一个迭代步
- `.save_model(filename, num_iteration=-1)` : 保存 `Booster` 对象到文件中。
  - 参数:
    - `filename` : 一个字符串, 给出了保存的文件的文件名
    - `num_iteration` : 一个整数, 指定需要保存的是第几轮训练的结果。如果小于0, 则最佳迭代步的结果 (如果存在的话) 将被保存。默认为 `-1`。
- `.set_attr(**kwargs)` : 设置 `Booster` 的属性。
  - 参数: `kwargs` : 关键字参数, 用于设定 `Booster` 属性。对于值为 `None` 的设置, 等效于删除该属性。
- `.set_network(machines, local_listen_port=12400, listen_time_out=120, num_machines=1)` : 配置网络
  - 参数:
    - `machines` : 一个字符串的列表、或者字符串的集合。它给出了每台机器的名字
    - `local_listen_port` : 一个整数, 默认为 `12400`, 指定了监听端口
    - `listen_time_out` : 一个整数, 默认为 `120`, 制定了 `socket` 超时的时间 (单位为分钟)
    - `num_machines` : 一个整数, 默认为 `1`, 表示并行学习的机器的数量
- `.set_train_data_name(name)` : 设置训练集的名字
  - 参数: `name` : 一个字符串, 表示训练集的名字
- `.update(train_set=None, fobj=None)` : 更新一个迭代步
  - 参数:
    - `train_set` : 一个 `Dataset` 或者 `None`, 表示训练集。如果为 `None`, 则上一个训练集被使用
    - `fobj` : 一个可调用对象或者 `None`, 表示自定义的目标函数。

注意: 如果是多类别分类任务, 则: `score` 首先根据 `class_id` 进行分组, 然后根据 `row_id` 分组。如果你想得到第 `i` 个样本在第 `j` 个类别上的得分, 访问方式为: `score[j*num_data+i]`。同理: `grad` 和 `hess` 也是以这样的方式访问。

- 返回值: 一个布尔值, 指示该次更新迭代步是否成功结束。

### 3. 示例:

```
_label_map={
    # 'Iris-setosa':0,
    'Iris-versicolor':0,
    'Iris-virginica':1
}
class BoosterTest:
```



```

def __init__(self):
    df = pd.read_csv('./data/iris.csv')
    _feature_names = ['Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width']
    x = df[_feature_names]
    y = df['Class'].map(lambda x: _label_map[x])

    train_X, test_X, train_Y, test_Y = train_test_split(x, y, test_size=0.3,
                                                         stratify=y, shuffle=True, random_state=1)
    print([item.shape for item in (train_X, test_X, train_Y, test_Y)])
    self._train_set = lgb.Dataset(data=train_X, label=train_Y,
                                   feature_name=_feature_names)
    self._validate_set = lgb.Dataset(data=test_X, label=test_Y,
                                       reference=self._train_set)

    self._booster = lgb.Booster(params={
        'boosting': 'gbdt',
        'verbosity': 1, # 打印消息
        'learning_rate': 0.1, # 学习率
        'num_leaves': 5,
        'max_depth': 5,
        'objective': 'binary',
        'metric': 'auc',
        'seed': 321,
    },
        train_set=self._train_set)
    self._booster.add_valid(self._validate_set, 'validate1')
    self._booster.set_train_data_name('trainAAAAA')

def print_attr(self):
    print('feature name:', self._booster.feature_name())
    # feature name: ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width']
    print('feature nums:', self._booster.num_feature())
    # feature nums: 4

def test_train(self):
    for i in range(0, 4):
        self._booster.update(self._train_set)
        print('after iter:%d'%self._booster.current_iteration())
        print('train eval:', self._booster.eval(self._train_set, name='train'))
        print('test eval:', self._booster.eval(self._validate_set, name='eval'))
    # after iter:1
    # train eval: [('train', 'auc', 0.9776530612244898, True)]
    # test eval: [('eval', 'auc', 0.9783333333333334, True)]
    # after iter:2
    # train eval: [('train', 'auc', 0.9907142857142858, True)]
    # test eval: [('eval', 'auc', 0.9872222222222222, True)]
    # after iter:3
    # train eval: [('train', 'auc', 0.9922448979591837, True)]
    # test eval: [('eval', 'auc', 0.9888888888888889, True)]
    # after iter:4
    # train eval: [('train', 'auc', 0.9922448979591837, True)]
    # test eval: [('eval', 'auc', 0.9888888888888889, True)]

def test(self):

```

```
self.print_attr()
self.test_train()
```

## 4.2.2 直接学习

1. `lightgbm.train()` 函数执行直接训练。

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

参数：

- `params`：一个字典，给出了训练参数
- `train_set`：一个 `Dataset` 对象，给出了训练集
- `num_boost_round`：一个整数，给出了 `boosting iteration` 的次数。默认为 `100`
- `valid_sets`：一个 `Dataset` 的列表或者 `None`，给出了训练期间用于 `evaluate` 的数据集。默认为 `None`
- `valid_names`：一个字符串列表或者 `None`，给出了 `valid_sets` 中每个数据集的名字。默认为 `None`
- `fobj`：一个可调用对象或者 `None`，表示自定义的目标函数。默认为 `None`
- `feval`：一个可调用对象或者 `None`，它表示自定义的 `evaluation` 函数。默认为 `None`。它的输入为 `(y_true, y_pred)`、或者 `(y_true, y_pred, weight)`、或者 `(y_true, y_pred, weight, group)`，返回一个元组：`(eval_name, eval_result, is_higher_better)`。或者返回该元组的列表。
- `init_model`：一个字符串或者 `None`，它给出了 `lightgbm model` 保存的文件名，或者 `Booster` 实例的名字。后续的训练在该 `model` 或者 `Booster` 实例的基础上继续训练。默认为 `None`
- `feature_name`：一个字符串列表或者 `'auto'`，它指定了特征的名字。默认为 `'auto'`
  - 如果数据源为 `pandas DataFrame` 并且 `feature_name='auto'`，则使用 `DataFrame` 的 `column names`
- `categorical_feature`：一个字符串列表、整数列表、或者 `'auto'`。它指定了 `categorical` 特征。默认为 `'auto'`
  - 如果是整数列表，则给定了 `categorical` 特征的下标
  - 如果是字符串列表，在给定了 `categorical` 特征的名字。此时必须设定 `feature_name` 参数。
  - 如果是 `'auto'` 并且数据源为 `pandas DataFrame`，则 `DataFrame` 的 `categorical` 列将作为 `categorical` 特征
- `early_stopping_rounds`：一个整数或者 `None`，表示验证集的 `score` 在连续多少轮未改善之后就早停。默认为 `None`

该参数要求至少有一个验证集以及一个 `metric`。

如果由多个验证集或者多个 `metric`，则对所有的验证集和所有的 `metric` 执行。

如果发生了早停，则模型会添加一个 `best_iteration` 字段。该字段持有了最佳的迭代步。

- `evals_result` : 一个字典或者 `None` , 这个字典用于存储在 `valid_sets` 中指定的所有验证集的所有验证结果。默认为 `None`
- `verbose_eval` : 一个布尔值或者整数。默认为 `True`
  - 如果是 `True` , 则在验证集上每个 `boosting stage` 打印对验证集评估的 `metric` 。
  - 如果是整数, 则每隔 `verbose_eval` 个 `boosting stage` 打印对验证集评估的 `metric` 。
  - 否则, 不打印这些

该参数要求至少由一个验证集。

- `learning_rates` : 一个列表、`None`、可调用对象。它指定了学习率。默认为 `None`
  - 如果为列表, 则它给出了每一个 `boosting` 步的学习率
  - 如果为一个可调用对象, 则在每个 `boosting` 步都调用它, 从而生成一个学习率
  - 如果为一个数值, 则学习率在学习期间都固定为它。

你可以使用学习率衰减从而生成一个更好的学习率序列。

- `keep_training_booster` : 一个布尔值, 指示训练得到的 `Booster` 对象是否还会继续训练。默认为 `False`
  - 如果为 `False` , 则返回的 `booster` 对象在返回之前将被转换为 `_InnerPredictor` 。

当然你也可以将 `_InnerPredictor` 传递给 `init_model` 参数从而继续训练。
- `callbacks` : 一个可调用对象的列表, 或者 `None` 。 它给出了在每个迭代步之后需要执行的函数。默认为 `None`

返回: 一个 `Booster` 实例

## 2. `lightgbm.cv()` 函数执行交叉验证训练。

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

参数:

- `params` : 一个字典, 给出了训练参数
- `train_set` : 一个 `Dataset` 对象, 给出了训练集
- `num_boost_round` : 一个整数, 给出了 `boosting iteration` 的次数。默认为 `10`
- `folds` : 一个生成器、一个迭代器、或者 `None` 。 如果是生成器或者迭代器, 则其迭代结果为元组: (训练部分样本下标列表, 测试部分样本下标列表), 分别给出了每个 `fold` 的训练部分和测试部分的下标。默认为 `None` 。

该参数比其它的拆分参数优先级更高。

- `nfold` : 一个整数, 指定了 `CV` 的数量。默认为 `5` 。
- `stratified` : 一个布尔值, 指示是否进行分层拆分。默认为 `True` 。
- `shuffle` : 一个布尔值, 指示是否在拆分之前先混洗数据。默认为 `True` 。

- `metrics` : 一个字符串、字符串列表、或者 `None` 。指定在 CV 过程中的 `evaluation metric` 。默认为 `None` 。

如果非 `None` , 则它会覆盖 `params` 的 `metric` 参数。

- `fobj` : 参考 `lightgbm.train()`
- `feval` : 参考 `lightgbm.train()`
- `init_model` : 参考 `lightgbm.train()`
- `feature_name` : 参考 `lightgbm.train()`
- `categorical_feature` : 参考 `lightgbm.train()`
- `early_stopping_rounds` : 一个整数或者 `None` , 表示 `CV error` 在连续多少轮未改善之后就早停。默认为 `None`

在返回的 `evaluation history` 中, 最后一项就是最佳迭代时的结果 ( 而不是最后一次迭代时的结果 ) 。

- `fpreproc` : 一个可调用对象或者 `None` , 默认为 `None` 。它是一个预处理函数, 在训练开始之前进行。

它的参数为 `(dtrain,dtest,params)` , 返回值是经过处理之后的 `(dtrain,dtest,params)`

- `verbose_eval` : 参考 `lightgbm.train()`
- `show_stdv` : 一个布尔值, 指示是否在训练过程中展示标准差信息。默认为 `True` 。

注意: 返回结果中始终包含标准差信息, 与该参数无关。

- `seed` : 一个整数, 给出了生成 `fold` 的随机数种子。默认为 `0`
- `callbacks` : 参考 `lightgbm.train()`

返回值: `evaluation history` , 它是一个字典, 格式为:

```
{
    'metric1-mean': [values], 'metric1-stdv': [values],
    'metric2-mean': [values], 'metric2-stdv': [values],
    ...
}
```

## 4.2.3 scikit-learn API

### 4.2.3.1 LGBMModel

1. `LGBMModel` 实现了 `lightgbm` 类似于 `scikit-learn` 的接口

```
class lightgbm.LGBMModel(boosting_type='gbdt', num_leaves=31, max_depth=-1,
    learning_rate=0.1, n_estimators=10, max_bin=255, subsample_for_bin=200000,
    objective=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20,
    subsample=1.0, subsample_freq=1, colsample_bytree=1.0, reg_alpha=0.0,
    reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, class_weight=None,
    **kwargs)
```

参数：

- `boosting_type`：一个字符串，指定了基学习器的算法。默认为 `'gbdt'`。可以为：
  - `'gbdt'`：表示传统的梯度提升决策树。默认值为 `'gbdt'`
  - `'rf'`：表示随机森林。
  - `'dart'`：表示带 `dropout` 的 `gbdt`
  - `goss`：表示 `Gradient-based One-Side Sampling` 的 `gbdt`
- `num_leaves`：一个整数，给出了一棵树上的叶子数。默认为 `31`
- `max_depth`：一个整数，限制了树模型的最大深度，默认值为 `-1`。
  - 如果小于0，则表示没有限制。
- `learning_rate`：一个浮点数，给出了学习率。默认为 `0.1`
- `n_estimators`：一个整数，给出了 `boosted trees` 的数量。默认为 `10`
- `max_bin`：一个整数，指定每个特征的最大分桶数量。默认为 `255`。
- `class_weight`：给出了每个类别的权重占比。
  - 可以为字符串 `'balanced'`，此时类别权重反比与类别的频率。
  - 可以为字典，此时人工给出了每个类别的权重。
  - 如果为 `None`，则认为每个类别的比例一样。

该参数仅用于多类分类问题。对于二类分类问题，可以使用 `is_unbalance` 参数。

- `subsample_for_bin`：一个整数，表示用来构建直方图的样本的数量。默认为 `200000`。
- `objective`：一个字符串、可调对象或者为 `None`，表示问题类型以及对应的目标函数。参考 2.2.1 核心参数->`objective`。

默认为 `None`，此时对于 `LGBMRegressor` 为 `'regression'`；对于 `LGBMClassifier` 为 `'binary'` 或者 `'multiclass'`；对于 `LGBMRanker` 为 `'lambda rank'`。

如果为自定义的可调对象，则它的签名为：`objective(y_true, y_pred) -> grad, hess`；或者签名为：`objective(y_true, y_pred, group) -> grad, hess`。其中：

- `y_true`：一个形状为 `(n_samples,)`（对于多类分类问题，则是 `(n_samples, n_classes)`）的 `array-like` 对象，给出了真实的标签值。
- `y_pred`：一个形状为 `(n_samples,)`（对于多类分类问题，则是 `(n_samples, n_classes)`）的 `array-like` 对象，给出了预测的标签值。
- `group`：一个 `array-like` 对象，给出了数据的分组信息。它用于 `ranking` 任务
- `grad`：一个形状为 `(n_samples,)`（对于多类分类问题，则是 `(n_samples, n_classes)`）的 `array-like` 对象，给出了每个样本的梯度值。
- `hess`：一个形状为 `(n_samples,)`（对于多类分类问题，则是 `(n_samples, n_classes)`）的 `array-like` 对象，给出了每个样本的二阶导数值。
- `min_split_gain`：一个浮点数，表示执行切分的最小增益，默认为0
- `min_child_weight`：一个浮点数，表示一个叶子节点上的最小 `hessian` 之和。（也就是叶节点样本权重之和的最小值）默认为 `1e-3`。
- `min_child_samples`：一个整数，表示一个叶子节点上包含的最少样本数量。默认值为 `20`
- `subsample`：一个浮点数，表示训练样本的采样比例。参考 2.2.2 学习控制参数->`subsample`。

- `subsample_freq` : 一个浮点数, 表示训练样本的采样频率。参考 2.2.2 学习控制参数->`subsample_freq`。
- `colsample_bytree` : 一个浮点数, 表示特征的采样比例。参考 2.2.2 学习控制参数->`colsample_bytree`。
- `reg_alpha` : 一个浮点数, 表示 L1 正则化系数。默认为 0
- `reg_lambda` : 一个浮点数, 表示 L2 正则化系数。默认为 0
- `random_state` : 一个整数或者 `None`, 表示随机数种子。如果为 `None`, 则使用默认的种子。默认为 `None`
- `n_jobs` : 一个整数, 指定并行的线程数量。如果为 -1, 则表示使用所有的 CPU。默认为 -1
- `silent` : 一个布尔值, 指示是否在训练过程中屏蔽输出。默认为 `True`。
- `kwargs` : 其它的参数。

## 2. 属性 :

- `.n_features_` : 一个整数, 给出了特征的数量
- `.classes_` : 一个形状为 `(n_classes,)` 的 `numpy array`, 给出了样本的标签。( 仅仅在分类问题中有效 )
- `.n_classes_` : 一个整数, 给出了类别的数量。( 仅仅在分类问题中有效 )
- `.best_score_` : 一个字典或者 `None`, 给出了训练完毕模型的最好的 `score`
- `.best_iteration_` : 一个字典或者 `None`。当 `early_stopping_round` 参数设定时, 它给出了训练完毕模型的最好的迭代步。
- `.objective_` : 一个字符串或者可调用对象, 给出了训练模型的目标函数
- `.booster_` : 一个 `Booster` 对象, 给出了底层的 `Booster` 对象。
- `.evals_result_` : 一个字典或者 `None`。当 `early_stopping_round` 参数设定时, 它给出了模型的 `evaluation results`。
- `.feature_importances_` : 一个形状为 `(n_features,)` 的 `numpy array`, 给出了特征的重要性 ( 值越大, 则对于的特征越重要 )。

## 3. 方法 :

- `.apply(X, num_iteration=0)` : 预测每个样本在每个树的哪个叶节点上。
  - 参数 :
    - `X` : 一个 `array-like` 对象, 或者一个 `sparse matrix`, 其形状为 `(n_samples, n_features)`, 表示测试样本集
    - `num_iteration` : 一个整数, 指示在预测时, 使用多少个子树。默认为 0, 表示使用所有的子树。
  - 返回值 : 一个 `array-like` 对象, 形状为 `(n_samples, n_trees)`。它给出了每个样本在每个子树的哪个节点上。
- `.fit()` : 训练模型。

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



参数：

- `X`：一个 `array-like` 对象，或者一个 `sparse matrix`，其形状为 `(n_samples, n_features)`，表示训练样本集
- `y`：一个 `array-like` 对象，形状为 `(n_samples,)`，给出了标签值。
- `sample_weight`：一个形状为 `(n_samples,)` 的 `array-like` 对象，或者为 `None`。给出了每个训练样本的权重。默认为 `None`
- `init_score`：一个形状为 `(n_samples,)` 的 `array-like` 对象，或者为 `None`。给出了每个训练样本的 `init score`。默认为 `None`
- `group`：一个形状为 `(n_samples,)` 的 `array-like` 对象，或者为 `None`。给出了每个训练样本的分组。默认为 `None`
- `eval_set`：一个元素为 `(X, y)` 的列表，或者 `None`。给出了验证集，用于早停。默认为 `None`。其中 `X, y` 的类型与参数 `X, y` 相同。
- `eval_names`：一个字符串的列表，或者 `None`。给出了每个验证集的名字。默认为 `None`。
- `eval_sample_weight`：一个 `array-like` 的列表，或者 `None`。给出了每个验证集中，每个样本的权重。默认为 `None`。
- `eval_init_score`：一个 `array-like` 的列表，或者 `None`。给出了每个验证集中，每个样本的 `init score`。默认为 `None`。
- `eval_group`：一个 `array-like` 的列表，或者 `None`。给出了每个验证集中，每个样本的分组。默认为 `None`。
- `eval_metric`：一个字符串、字符串列表、可调用对象、或者 `None`。给出了验证的 `metric`。默认为 `None`。
- `early_stopping_rounds`：一个整数或者 `None`，默认为 `None`。参考 `lightgbm.train()`
- `verbose`：一个布尔值。如果为 `True`，并且至少有一个验证集，则输出 `evaluation` 信息。
- `feature_name`：一个字符串列表、或者 `'auto'`。参考 `lightgbm.train()`
- `categorical_feature`：一个字符串列表、整数、或者 `'auto'`。参考 `lightgbm.train()`
- `callbacks`：一个可调用对象的列表或者为 `None`。参考 `lightgbm.train()`

返回值：`self`，即当前 `LGBMModel` 对象自己

- `.predict(X, raw_score=False, num_iteration=0)`：执行预测。

参数：

- `X`：一个 `array-like` 对象，或者一个 `sparse matrix`，其形状为 `(n_samples, n_features)`，表示测试样本集。  
注意：如果是 `numpy array` 或者 `pandas dataframe` 时，要求数据的列必须与训练时的列顺序一致。
- `raw_score`：一个布尔值，指示是否输出 `raw score`。默认为 `False`
- `num_iteration`：一个整数，指示在预测时，使用多少个子树。默认为 `0`，表示使用所有的子树。

返回值：一个形状为 `(n_samples,)` 或者形状为 `(n_samples,n_classed)` 的 `array-like` 对象，表示预测结果

#### 4.2.3.2 LGBMClassifier

1. `LGBMClassifier` 是 `LGBMModel` 的子类，它用于分类任务。

```
class lightgbm.LGBMClassifier(boosting_type='gbdt', num_leaves=31, max_depth=-1,
                               learning_rate=0.1, n_estimators=10, max_bin=255, subsample_for_bin=200000,
                               objective=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20,
                               subsample=1.0, subsample_freq=1, colsample_bytree=1.0, reg_alpha=0.0,
                               reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, **kwargs)
```

参数：参考 `LGBMModel`

2. 属性：参考 `LGBMModel`

3. 方法：

- `.fit()`：训练模型

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

参数：参考 `LGBMModel.fit()`

返回值：参考 `LGBMModel.fit()`

- `.predict_proba(X, raw_score=False, num_iteration=0)`：预测每个样本在每个类上的概率。

参数：参考 `LGBMModel.predict()`

返回值：一个形状为 `(n_samples,n_classes)` 的 `array-like` 对象，给出了每个样本在每个类别上的概率。

- 其它方法参考 `LGBMModel`

#### 4.2.3.3 LGBMRegressor

1. `LGBMRegressor` 是 `LGBMModel` 的子类，它用于回归任务。

```
class lightgbm.LGBMRegressor(boosting_type='gbdt', num_leaves=31, max_depth=-1,
                              learning_rate=0.1, n_estimators=10, max_bin=255, subsample_for_bin=200000,
                              objective=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20,
                              subsample=1.0, subsample_freq=1, colsample_bytree=1.0, reg_alpha=0.0,
                              reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, **kwargs)
```

参数：参考 `LGBMModel`

2. 属性：参考 `LGBMModel`

3. 方法：

- `.fit()` : 训练模型

```
fit(X, y, sample_weight=None, init_score=None, eval_set=None, eval_names=None,
    eval_sample_weight=None, eval_init_score=None, eval_metric='l2',
    early_stopping_rounds=None, verbose=True, feature_name='auto',
    categorical_feature='auto', callbacks=None)
```

参数：参考 `LGBMModel.fit()`

返回值：参考 `LGBMModel.fit()`

- 其它方法参考 `LGBMModel`

#### 4.2.3.4 LGBMRanker

1. `LGBMRanker` 是 `LGBMModel` 的子类，它用于 `ranking` 任务。

```
class lightgbm.LGBMRanker(boosting_type='gbdt', num_leaves=31, max_depth=-1,
    learning_rate=0.1, n_estimators=10, max_bin=255, subsample_for_bin=200000,
    objective=None, min_split_gain=0.0, min_child_weight=0.001, min_child_samples=20,
    subsample=1.0, subsample_freq=1, colsample_bytree=1.0, reg_alpha=0.0,
    reg_lambda=0.0, random_state=None, n_jobs=-1, silent=True, **kwargs)
```

参数：参考 `LGBMModel`

2. 属性：参考 `LGBMModel`

3. 方法：

- `.fit()` : 训练模型

```
fit(X, y, sample_weight=None, init_score=None, group=None, eval_set=None,
    eval_names=None, eval_sample_weight=None, eval_init_score=None,
    eval_group=None, eval_metric='ndcg', eval_at=[1], early_stopping_rounds=None,
    verbose=True, feature_name='auto', categorical_feature='auto', callbacks=None)
```

参数：

- `eval_at` : 一个整数列表，给出了 `NDCG` 的 `evaluation position`。默认为 `[1]`。
- 其它参数参考 `LGBMModel.fit()`

返回值：参考 `LGBMModel.fit()`

- 其它方法参考 `LGBMModel`

#### 4.2.3.5 Callbacks

1. 这里介绍的 `callback` 生成一些可调用对象，它们用于 `LGBMModel.fit()` 方法的 `callbacks` 参数。
2. `lightgbm.early_stopping(stopping_rounds, verbose=True)` : 创建一个回调函数，它用于触发早停。

触发早停时，要求至少由一个验证集以及至少有一种评估指标。如果由多个，则将它们都检查一遍。

参数：

- `stopping_rounds` : 一个整数。如果一个验证集的度量在 `early_stopping_round` 循环中没有提升, 则停止训练。如果为0则表示不开启早停。
- `verbose` : 一个布尔值。是否打印早停的信息。

返回值: 一个回调函数。

3. `lightgbm.print_evaluation(period=1, show_stdv=True)` : 创建一个回调函数, 它用于打印 `evaluation` 的结果。

参数:

- `period` : 一个整数, 默认为1。给出了打印 `evaluation` 的周期。默认每个周期都打印。
- `show_stdv` : 一个布尔值, 默认为 `True`。指定是否打印标准差的信息 (如果提供了标准差的话)。

返回值: 一个回调函数。

4. `lightgbm.record_evaluation(eval_result)` : 创建一个回调函数, 它用于将 `evaluation history` 写入到 `eval_result` 中。

参数: `eval_result` : 一个字典, 它将用于存放 `evaluation history`。

返回值: 一个回调函数。

5. `lightgbm.reset_parameter(**kwargs)` : 创建一个回调函数, 它用于在第一次迭代之后重新设置参数。

注意: 当第一次迭代时, 初始的参数仍然发挥作用。

参数:

- `kwargs` : 一些关键字参数 (如 `key=val`), 每个关键字参数的值必须是一个列表或者一个函数。给出了每一个迭代步的相应参数。
  - 如果是列表, 则 `current_round` 迭代时的参数为: `val[current_round]`
  - 如果是函数, 则 `current_round` 迭代时的参数值为: `val(current_round)`

返回值: 一个回调函数。

## 4.4 绘图API

1. `lightgbm.plot_importance()` : 绘制特征的重要性。

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

参数:

- `booster` : 一个 `Booster` 或者 `LGBMModel` 对象。即将绘制的就是该对象的特征的重要性
- `ax` : 一个 `matplotlib.axes.Axes` 实例或者 `None`。它制定了绘制图形的 `Axes`。
 

如果为 `None`, 则创建一个新的 `figure` 以及 `axes`。默认为 `None`。
- `height` : 一个浮点数, 给出了 `bar` 的高度 (将被传递给 `ax.barh()` 方法)。默认为 `0.2`
- `xlim` : 一个二元元组或者 `None`, 给出了 `x` 轴的范围 (将被传递给 `ax.xlim()` 方法)。默认为 `None`

- `ylim` : 一个二元元组或者 `None` , 给出了 `y` 轴的范围 ( 将被传递给 `ax.ylim()` 方法 ) 。 默认为 `None` 。
- `title` : 一个字符串或者 `None` , 给出了 `Axes` 的标题。默认为 `Feature importance` 。  
如果为 `None` , 则没有标题。
- `xlabel` : 一个字符串或者 `None` , 给出了 `x` 轴的标题。默认为 `Feature importance` 。  
如果为 `None` , 则没有标题。
- `ylabel` : 一个字符串或者 `None` , 给出了 `Y` 的标题。默认为 `Features` 。  
如果为 `None` , 则没有标题。
- `importance_type` : 一个字符串, 给出了如何计算出重要性的。默认为 `'split'` 。 参考 `lightgbm.Booster.feature_importance()` 方法
- `max_num_features` : 一个整数或者 `None` , 给出了最多展示多少个特征的重要性。默认为 `None` 。  
如果为 `None` 或者小于 `1` 的整数, 则展示所有的。
- `ignore_zero` : 一个布尔值, 指定是否忽略为 `0` 的特征。默认为 `True` 。
- `figsize` : 一个二元的元组或者 `None` , 指定了图像的尺寸。默认为 `None` 。
- `grid` : 一个布尔值, 指示是否添加网格。默认为 `True` 。
- `kwargs` : 其它的关键字参数。它将被传递给 `ax.barh()` 方法。

返回值: 一个 `matplotlib.axes.Axes` 对象, 它就是传入的 `ax` 本身。

## 2. `lightgbm.plot_metric()` : 在训练过程中绘制一个 `metric`

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

参数:

- `booster` : 一个字典或者一个 `LGBMModel` 实例。
  - 如果是一个字典, 则它是由 `lightgbm.train()` 返回的字典
- `metric` : 一个字符串或者 `None` , 指定了要绘制的 `metric` 的名字。如果为 `None` , 则从字典中取出第一个 `metric` ( 根据 `hashcode` 的顺序 ) 。默认为 `None` 。  
只支持绘制一个 `metric` , 因为不同的 `metric` 无法绘制在一张图上 ( 不同的 `metric` 有不同的量级 ) 。
- `dataset_names` : 一个字符串列表, 或者 `None` 。 它给出了用于计算 `metric` 的样本集的名字。如果为 `None` , 则使用所有的样本集。默认为 `None` 。
- `title` : 一个字符串或者 `None` , 给出了 `Axes` 的标题。默认为 `Metric during training` 。  
如果为 `None` , 则没有标题。
- `xlabel` : 一个字符串或者 `None` , 给出了 `x` 轴的标题。默认为 `Iterations` 。  
如果为 `None` , 则没有标题。
- `ylabel` : 一个字符串或者 `None` , 给出了 `Y` 的标题。默认为 `auto` 。

- 如果为 `None` , 则没有标题。
- 如果为 `'auto'` , 则使用 `metric` 的名字。
- 其它参数: 参考 `lightgbm.plot_importance()`

返回值: 一个 `matplotlib.axes.Axes` 对象, 它就是传入的 `ax` 本身。

### 3. `lightgbm.plot_tree()` : 绘制指定的树模型。

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

参数:

- `booster` : 一个 `Booster` 或者 `LGBMModel` 对象。即将绘制的就是该对象的树模型。
- `tree_index` : 一个整数, 指定要绘制哪棵树。默认为 `0`。
- `graph_attr` : 一个字典或者 `None` , 给出了 `graphviz graph` 的属性。默认为 `None`
- `node_attr` : 一个字典或者 `None` , 给出了 `graphviz node` 的属性。默认为 `None`
- `edge_attr` : 一个字典或者 `None` , 给出了 `graphviz edge` 的属性。默认为 `None`
- `show_info` : 一个列表或者 `None` , 给出了将要在 `graph node` 中显示哪些信息。可以为: `'split_gain', 'internal_value', 'internal_count', 'leaf_count'` 。默认为 `None`。
- 其它参数: 参考 `lightgbm.plot_importance()`

返回值: 一个 `matplotlib.axes.Axes` 对象, 它就是传入的 `ax` 本身。

### 4. `lightgbm.create_tree_digraph()` : 绘制指定的树模型, 但是返回一个 `digraph` , 而不是直接绘制。

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

参数:

- `booster` : 一个 `Booster` 或者 `LGBMModel` 对象。即将绘制的就是该对象的树模型。
- `name` : 一个字符串或者 `None` 。给出了 `graphviz` 源文件的名字。默认为 `None`。
- `comment` : 一个字符串或者 `None` 。给出了添加到 `graphviz` 源文件第一行的评论。默认为 `None`。
- `filename` : 一个字符串或者 `None` 。给出了保存 `graphviz` 源文件的名字。如果为 `None` , 则是 `name+'.gv'` 。默认为 `None`
- `directory` : 一个字符串或者 `None` , 给出了保存和渲染 `graphviz` 文件的目录。默认为 `None`
- `format` : 一个字符串或者 `None` , 表示输出图片的格式。可以为 `'png', 'pdf', ...` 。默认为 `None`
- `engine` : 一个字符串或者 `None` , 制定了 `graphviz` 的排版引擎。可以为 `'dot', 'neato', ...` 。默认为 `None`
- `encoding` : 一个字符串或者 `None` , 指定了 `graphviz` 源文件的编码。默认为 `None`
- `body` : 一个字符串列表或者 `None` , 给出了添加到 `graphviz graph body` 中的线条。默认为 `None`
- `strict` : 一个布尔值, 指示是否应该合并 `multi-edges` 。默认为 `False`。
- 其它参数: 参考 `lightgbm.plot_tree()`

返回值: 一个 `graphviz.Digraph` 对象, 代表指定的树模型的 `digraph` 。



## 4.5 Booster API 转换

1. 从 `LGBMModel` 转换到 `Booster` : 通过 `.booster_` 属性来获取底层的 `Booster`。

源码：

```
@property
def booster_(self):
    """Get the underlying lightgbm Booster of this model."""
    if self._Booster is None:
        raise LGBMNotFittedError('No booster found. Need to call fit beforehand.')
    return self._Booster
```

- 用途：当使用 `scikit-learn API` 学习到一个模型之后，需要保存模型。则需要先转换成 `Booster` 对象，再调用其 `.save_model()` 方法。
- 2. 使用 `Booster` 来预测分类的概率：
  - 因为 `Booster` 仅仅提供了 `predict` 接口，而未提供 `predict_proba` 接口。因此需要使用这种转换
  - 在 `LGBMClassifier` 的 `predict_proba` 方法中的源码：

```
class_probs = self.booster_.predict(X, raw_score=raw_score,
num_iteration=num_iteration)
if self._n_classes > 2: return class_probs
else: return np.vstack((1. - class_probs, class_probs)).transpose()
```

## 五、Docker

### 5.1 安装和使用

1. `cli` 模式：

- 安装：

```
mkdir lightgbm-docker
cd lightgbm-docker
wget https://raw.githubusercontent.com/Microsoft/LightGBM/master/docker/dockerfile-cli
docker build -t lightgbm-cli -f dockerfile-cli .
```

- 使用：

```
docker run --rm -it \  
--volume $HOME/lgbm.conf:/lgbm.conf \  
--volume $HOME/model.txt:/model.txt \  
--volume $HOME/tmp:/out \  
lightgbm-cli \  
config=lgbm.conf
```

其中 `config` 给出了模型的参数。

## 2. python 模式：

### ◦ 安装：

```
mkdir lightgbm-docker  
cd lightgbm-docker  
wget https://raw.githubusercontent.com/Microsoft/LightGBM/master/docker/dockerfile-  
python  
docker build -t lightgbm -f dockerfile-python .
```

### ◦ 使用：

```
docker run --rm -it lightgbm
```