

Xgboost使用

一、安装

1. 安装步骤:

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

```
git clone --recursive https://github.com/dmlc/xgboost
# 对于 windows 用户, 需要执行:
# git submodule init
# git submodule update
cd xgboost
make -j4
```

如果需要支持 `GPU`, 则执行下面的编译步骤:

```
cd xgboost
mkdir build
cd build
cmake .. -DUSE_CUDA=ON
cd ..
make -j4
```

- 安装 `python` 支持

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

二、调参

2.1 调参指导

1. 当出现过拟合时, 有两类参数可以缓解:

- 第一类参数: 用于直接控制模型的复杂度。包括 `max_depth`, `min_child_weight`, `gamma` 等参数
- 第二类参数: 用于增加随机性, 从而使得模型在训练时对于噪音不敏感。包括 `subsample`, `colsample_bytree`

你也可以直接减少步长 `eta`, 但是此时需要增加 `num_round` 参数。

2. 当遇到数据不平衡时 (如广告点击率预测任务), 有两种方式提高模型的预测性能:

- 如果你关心的是预测的 `AUC` :
 - 你可以通过 `scale_pos_weight` 参数来平衡正负样本的权重
 - 使用 `AUC` 来评估

- 如果你关心的是预测的正确率：
 - 你不能重新平衡正负样本
 - 设置 `max_delta_step` 为一个有限的值（如 1），从而有助于收敛

2.2 参数

2.2.1. 通用参数

1. `booster`：指定了使用那一种 `booster`。可选的值为：
 - `'gbtree'`：表示采用 `xgboost`（默认值）
 - `'gblinear'`：表示采用线性模型。
`'gblinear'` 使用带 `l1, l2` 正则化的线性回归模型作为基学习器。因为 `boost` 算法是一个线性叠加的过程，而线性回归模型也是一个线性叠加的过程。因此叠加的最终结果就是一个整体的线性模型，`xgboost` 最后会获得这个线性模型的系数。
 - `'dart'`：表示采用 `dart booster`
2. `silent`：如果为 0（默认值），则表示打印运行时的信息；如果为 1，则表示 `silent mode`（不打印这些信息）
3. `nthread`：指定了运行时的并行线程的数量。如果未设定该参数，则默认值为可用的最大线程数。
4. `num_pbuffer`：指定了 `prediction buffer` 的大小。通常设定为训练样本的数量。该参数由 `xgboost` 自动设定，无需用户指定。
 该 `buffer` 用于保存上一轮 `boosting step` 的预测结果。
5. `num_feature`：样本的特征数量。通常设定为特征的最大维数。该参数由 `xgboost` 自动设定，无需用户指定。

2.2.2 tree booster 参数

针对 `tree booster` 的参数（适用于 `booster=gbtree, dart`）：

1. `eta`：也称作学习率。默认为 0.3。范围为 `[0, 1]`
2. `gamma`：也称作最小划分损失 `min_split_loss`。它刻画的是：对于一个叶子节点，当对它采取划分之后，损失函数的降低值的阈值。
 - 如果大于该阈值，则该叶子节点值得继续划分
 - 如果小于该阈值，则该叶子节点不值得继续划分
 该值越大，则算法越保守（尽可能的少划分）。默认值为 0
3. `max_depth`：每棵子树的最大深度。其取值范围为 `[0, ∞]`，0 表示没有限制，默认值为 6。
 该值越大，则子树越复杂；值越小，则子树越简单。
4. `min_child_weight`：子节点的权重阈值。它刻画的是：对于一个叶子节点，当对它采取划分之后，它的所有子节点的权重之和的阈值。
 - 如果它的所有子节点的权重之和大于该阈值，则该叶子节点值得继续划分
 - 如果它的所有子节点的权重之和小于该阈值，则该叶子节点不值得继续划分

所谓的权重：

- 对于线性模型（`booster=gblinear`），权重就是：叶子节点包含的样本数量。因此该参数就是每个节点包含的最少样本数量。

- 对于树模型 (`booster=gbt, dart`), 权重就是: 叶子节点包含样本的所有二阶偏导数之和。

该值越大, 则算法越保守 (尽可能的少划分)。默认值为 1

5. `max_delta_step`: 每棵树的权重估计时的最大 `delta step`。取值范围为 $[0, \infty]$, 0 表示没有限制, 默认值为 0。

通常该参数不需要设置, 但是在逻辑回归中, 如果类别比例非常不平衡时, 该参数可能有帮助。

6. `subsample`: 对训练样本的采样比例。取值范围为 $(0, 1]$, 默认值为 1。

如果为 0.5, 表示随机使用一半的训练样本来训练子树。它有助于缓解过拟合。

7. `colsample_bytree`: 构建子树时, 对特征的采样比例。取值范围为 $(0, 1]$, 默认值为 1。

如果为 0.5, 表示随机使用一半的特征来训练子树。它有助于缓解过拟合。

8. `colsample_bylevel`: 寻找划分点时, 对特征的采样比例。取值范围为 $(0, 1]$, 默认值为 1。

如果为 0.5, 表示随机使用一半的特征来寻找最佳划分点。它有助于缓解过拟合。

9. `lambda`: L2 正则化系数 (基于 `weights` 的正则化), 默认为 1。该值越大则模型越简单

10. `alpha`: L1 正则化系数 (基于 `weights` 的正则化), 默认为 0。该值越大则模型越简单

11. `tree_method`: 指定了构建树的算法, 可以为下列的值: (默认为 'auto')

- 'auto': 使用启发式算法来选择一个更快的 `tree_method`:
 - 对于小的和中等的训练集, 使用 `exact greedy` 算法分裂节点
 - 对于非常大的训练集, 使用近似算法分裂节点
 - 旧版本在单机上总是使用 `exact greedy` 分裂节点
- 'exact': 使用 `exact greedy` 算法分裂节点
- 'approx': 使用近似算法分裂节点
- 'hist': 使用 `histogram` 优化的近似算法分裂节点 (比如使用了 `bin cacheing` 优化)
- 'gpu_exact': 基于 GPU 的 `exact greedy` 算法分裂节点
- 'gpu_hist': 基于 GPU 的 `histogram` 算法分裂节点

注意: 分布式, 以及外存版本的算法只支持 'approx', 'hist', 'gpu_hist' 等近似算法

12. `sketch_eps`: 指定了分桶的步长。其取值范围为 $(0, 1)$, 默认值为 0.03。

它仅仅用于 `tree_method='approx'`。

它会产生大约 $\frac{1}{\text{sketch_eps}}$ 个分桶。它并不会显示的分桶, 而是会每隔 `sketch_eps` 个单位 (一个单位表示最大值减去最小值的区间) 统计一次。

用户通常不需要调整该参数。

13. `scale_pos_weight`: 用于调整正负样本的权重, 常用于类别不平衡的分类问题。默认为 1。

一个典型的参数值为: `负样本数量/正样本数量`

14. `updater`: 它是一个逗号分隔的字符串, 指定了一组需要运行的 `tree updaters`, 用于构建和修正决策树。默认为 'grow_colmaker,prune'。

该参数通常是自动设定的, 无需用户指定。但是用户也可以显式的指定。

15. `refresh_leaf`: 它是一个 `updater plugin`。如果为 `true`, 则树节点的统计数据 and 树的叶节点数据都被更新; 否则只有树节点的统计数据被更新。默认为 1

16. `process_type` : 指定要执行的处理过程 (如: 创建子树、更新子树)。该参数通常是自动设定的, 无需用户指定。
17. `grow_policy` : 用于指定子树的生长策略。仅仅支持 `tree_method='hist'`。有两种策略:
 - o `'depthwise'` : 优先拆分那些靠近根部的子节点。默认为 `'depthwise'`
 - o `'lossguide'` : 优先拆分导致损失函数降低最快的子节点
18. `max_leaves` : 最多的叶子节点。如果为0, 则没有限制。默认值为 0。
该参数仅仅和 `grow_policy='lossguide'` 关系较大。
19. `max_bin` : 指定了最大的分桶数量。默认值为 256。
该参数仅仅当 `tree_method='hist','gpu_hist'` 时有效。
20. `predictor` : 指定预测器的算法, 默认为 `'cpu_predictor'`。可以为:
 - o `'cpu_predictor'` : 使用 CPU 来预测
 - o `'gpu_predictor'` : 使用 GPU 来预测。对于 `tree_method='gpu_exact,gpu_hist'`, `'gpu_predictor'` 是默认值。

2.2.3 dart booster 参数

1. `sample_type` : 它指定了丢弃时的策略:
 - o `'uniform'` : 随机丢弃子树 (默认值)
 - o `'weighted'` : 根据权重的比例来丢弃子树
2. `normaliz_type` : 它指定了归一化策略:
 - o `'tree'` : 新的子树将被缩放为: $\frac{1}{K+\nu}$; 被丢弃的子树被缩放为 $\frac{\nu}{K+\nu}$ 。其中 ν 为学习率, K 为被丢弃的子树的数量
 - o `'forest'` : 新的子树将被缩放为: $\frac{1}{1+\nu}$; 被丢弃的子树被缩放为 $\frac{\nu}{1+\nu}$ 。其中 ν 为学习率
3. `rate_drop` : `dropout rate`, 指定了当前要丢弃的子树占当前所有子树的比例。范围为 `[0.0,1.0]`, 默认为 `0.0`。
4. `one_drop` : 如果该参数为 `true`, 则在 `dropout` 期间, 至少有一个子树总是被丢弃。默认为 0。
5. `skip_drop` : 它指定了不执行 `dropout` 的概率, 其范围是 `[0.0,1.0]`, 默认为 0.0。
如果跳过了 `dropout`, 则新的子树直接加入到模型中 (和 `xgboost` 相同的方式)

2.2.4 linear booster 参数

1. `lambda` : `L2` 正则化系数 (基于 `weights` 的正则化), 默认为 0。该值越大则模型越简单
2. `alpha` : `L1` 正则化系数 (基于 `weights` 的正则化), 默认为 0。该值越大则模型越简单
3. `lambda_bias` : `L2` 正则化系数 (基于 `bias` 的正则化), 默认为 0。
没有基于 `bias` 的 `L1` 正则化, 因为它不重要。

2.4.5 tweedie regression 参数:

1. `tweedie_variance_power` : 指定了 `tweedie` 分布的方差。取值范围为 `(1,2)`, 默认为 1.5。
越接近1, 则越接近泊松分布; 越接近2, 则越接近 `gamma` 分布。

2.4.6 学习任务参数

1. `objective` : 指定任务类型, 默认为 `'reg:linear'`。

- `'reg:linear'`: 线性回归模型。它的模型输出是连续值
- `'reg:logistic'`: 逻辑回归模型。它的模型输出是连续值, 位于区间 `[0,1]`。
- `'binary:logistic'`: 二分类的逻辑回归模型, 它的模型输出是连续值, 位于区间 `[0,1]`, 表示取正负类别的概率。
它和 `'reg:logistic'` 几乎完全相同, 除了有一点不同:
 - `'reg:logistic'` 的默认 `evaluation metric` 是 `rmse`。
 - `'binary:logistic'` 的默认 `evaluation metric` 是 `error`
- `'binary:logitraw'`: 二分类的逻辑回归模型, 输出为分数值 (在 `logistic` 转换之前的值)
- `'count:poisson'`: 对 `count data` 的 `poisson regression`, 输出为泊松分布的均值。
- `'multi:softmax'`: 基于 `softmax` 的多分类模型。此时你需要设定 `num_class` 参数来指定类别数量。
- `'multi:softprob'`: 基于 `softmax` 的多分类模型, 但是它的输出是一个矩阵: `ndata*nclass`, 给出了每个样本属于每个类别的概率。
- `'rank:pairwise'`: 排序模型 (优化目标为最小化 `pairwise loss`)
- `'reg:gamma'`: `gamma regression`, 输出为伽马分布的均值。
- `'reg:tweedie'`: `'tweedie regression'`。

2. `base_score`: 所有样本的初始预测分, 它用于设定一个初始的、全局的 `bias`。默认为 0.5。

- 当迭代的数量足够大时, 该参数没有什么影响

3. `eval_metric`: 用于验证集的评估指标。其默认值和 `objective` 参数高度相关。

回归问题的默认值是 `rmse`; 分类问题的默认值是 `error`; 排序问题的默认值是 `mean average precision`。

你可以指定多个 `evaluation metrics`。

如果有多个验证集, 以及多个评估指标, 则: 使用最后一个验证集的最后一个评估指标来做早停。但是还是会计算出所有的验证集的所有评估指标。

- `'rmse'`: 均方误差。
- `'mae'`: 绝对值平均误差
- `'logloss'`: 负的对数似然函数
- `'error'`: 二分类的错误率。它计算的是: `预测错误的样本数/所有样本数`

所谓的预测是: 正类概率大于0.5的样本预测为正类; 否则为负类 (即阈值为 0.5)

- `'error@t'`: 二分类的错误率。但是它的阈值不再是 0.5, 而是由字符串 `t` 给出 (它是一个数值转换的字符串)
- `'merror'`: 多类分类的错误率。它计算的是: `预测错误的样本数/所有样本数`
- `'mlogloss'`: 多类分类的负对数似然函数
- `'auc'`: `AUC` 得分
- `'ndcg'`: `Normalized Discounted Cumulative Gain` 得分
- `'map'`: `Mean average precision` 得分
- `'ndcg@n', 'map@n'`: `n` 为一个整数, 用于切分验证集的 `top` 样本来求值。

- 'ndcg-', 'map-', 'ndcg@n-', 'map@n-': NDCG and MAP will evaluate the score of a list without any positive samples as 1. By adding "-" in the evaluation metric XGBoost will evaluate these score as 0 to be consistent under some conditions. training repeatedly
 - poisson-nloglik: 对于泊松回归, 使用负的对数似然
 - gamma-nloglik: 对于伽马回归, 使用负的对数似然
 - gamma-deviance: 对于伽马回归, 使用残差的方差
 - tweedie-nloglik: 对于 tweedie 回归, 使用负的对数似然
4. seed: 随机数种子, 默认为 0。

三、外存计算

1. 对于 external-memory 和 in-memory 计算, 二者几乎没有区别。除了在文件名上有所不同。
- in-memory 的文件名为: filename
 - external-memory 的文件名为: filename#cacheprefix。其中:
 - filename: 是你想加载的数据集 (libsvm 文件) 的路径名
当前只支持导入 libsvm 格式的文件
 - cacheprefix: 指定的 cache 文件的路径名。 xgboost 将使用它来做 external memory cache

如:

```
dtrain = xgb.DMatrix('../data/my_data.txt.train#train_cache.cache')
```

此时你会发现 my_data.txt 所在的位置会由 xgboost 创建一个 my_cache.cache 文件。

2. 推荐将 nthread 设置为真实 CPU 的数量。
- 现代的 CPU 都支持超线程, 如 4核8线程。此时 nthread 设置为 4 而不是 8
3. 对于分布式计算, 外存计算时文件名的设定方法也相同:

```
data = "hdfs:///path-to-data/my_data.txt.train#train_cache.cache"
```

四、GPU计算

1. xgboost 支持使用 gpu 计算, 前提是安装时开启了 GPU 支持
2. 要想使用 GPU 训练, 需要指定 tree_method 参数为下列的值:
- 'gpu_exact': 标准的 xgboost 算法。它会对每个分裂点进行精确的搜索。相对于 'gpu_hist', 它的训练速度更慢, 占用更多内存
 - 'gpu_hist': 使用 xgboost histogram 近似算法。它的训练速度更快, 占用更少内存
3. 当 tree_method 为 'gpu_exact', 'gpu_hist' 时, 模型的 predict 默认采用 GPU 加速。

你可以通过设置 predictor 参数来指定 predict 时的计算设备:

- 'cpu_predictor': 使用 CPU 来执行模型预测

- 'gpu_predictor': 使用 GPU 来执行模型预测
4. 多 GPU 可以通过 grow_gpu_hist 参数和 n_gpus 参数配合使用。
- 可以通过 gpu_id 参数来选择设备，默认为 0。如果非 0，则 GPU 的编号规则为 $\text{mod}(\text{gpu_id} + i) \% n_visible_devices$ for i in $0 \sim n_gpus - 1$
 - 如果 n_gpus 设置为 -1，则所有的 GPU 都被使用。它默认为 1。
5. 多 GPU 不一定比单个 GPU 更快，因为 PCI 总线的带宽限制，数据传输速度可能成为瓶颈。
6. GPU 计算支持的参数：

parameter	gpu_exact	gpu_hist
subsample	X	✓
colsample_bytree	X	✓
colsample_bylevel	X	✓
max_bin	X	✓
gpu_id	✓	✓
n_gpus	X	✓
predictor	✓	✓
grow_policy	X	✓
monotone_constraints	X	✓

五、单调约束

1. 在模型中可能会有一些单调的约束：当 $x \leq x'$ 时：
- 若 $f(x_1, x_2, \dots, x, \dots, x_n) \leq f(x_1, x_2, \dots, x', \dots, x_n)$ ，则称该约束为单调递增约束
 - 若 $f(x_1, x_2, \dots, x, \dots, x_n) \geq f(x_1, x_2, \dots, x', \dots, x_n)$ ，则称该约束为单调递减约束
2. 如果想在 xgboost 中添加单调约束，则可以设置 monotone_constraints 参数。

假设样本有 2 个特征，则：

- `params['monotone_constraints'] = "(1,-1)"`：表示第一个特征是单调递增；第二个特征是单调递减
- `params['monotone_constraints'] = "(1,0)"`：表示第一个特征是单调递增；第二个特征没有约束
- `params['monotone_constraints'] = "(1,1)"`：表示第一个特征是单调递增；第二个特征是单调递增

右侧的 1 表示单调递增约束；0 表示无约束；-1 表示单调递减约束。有多少个特征，就对应多少个数值。

六、DART booster

1. 在 GBDT 中，越早期加入的子树越重要；越后期加入的子树越不重要。
2. DART booster 原理：为了缓解过拟合，采用 dropout 技术，随机丢弃一些树。
3. 由于引入了随机性，因此 dart 和 gbtrees 有以下不同：

- 训练速度更慢
- 早停不稳定

4. `DART booster` 也是使用与提升树相同的前向分步算法

- 第 m 步, 假设随机丢弃 K 棵, 被丢弃的树的下标为集合 \mathbb{K} 。

令 $D = \sum_{k \in \mathbb{K}} h_k$, 第 m 棵树为 h_m 。则目标函数为:

$$\mathcal{L} = \sum_{i=1}^N L(y_i, \hat{y}_i^{<m-1>} - D(\mathbf{x}_i) + \nu h_m(\mathbf{x}_i)) + \Omega(f_m)。$$

- 由于 `dropout` 在设定目标函数时引入了随机丢弃, 因此如果直接引入 h_m , 则会引起超调。因此引入缩放因子, 这称作为归一化: $f_m = \sum_{k \neq \mathbb{K}} h_k + \alpha (\sum_{k \in \mathbb{K}} h_k + b \nu h_m)$ 。

- 其中 b 为新的子树与丢弃的子树的权重之比, α 为修正因子。
- 令 $\hat{M} = \sum_{k \neq \mathbb{K}} h_k$ 。采用归一化的原因是: h_m 试图缩小 \hat{M} 到目标之间的 `gap`; 而 D 也会试图缩小 \hat{M} 到目标之间的 `gap`。

如果同时引入随机丢弃的子树集合 D , 以及新的子树 h_m , 则会引起超调。

- 有两种归一化策略:
 - `'tree'`: 新加入的子树具有和每个丢弃的子树一样的权重, 假设都是 $\frac{1}{K}$ 。

此时 $b = \frac{1}{K}$, 则有:

$$\alpha \left(\sum_{k \in \mathbb{K}} h_k + b \nu h_m \right) = \alpha \left(\sum_{k \in \mathbb{K}} h_k + \frac{\nu}{K} h_m \right) \sim \alpha \left(1 + \frac{\nu}{K} \right) D = \alpha \frac{K + \nu}{K} D$$

要想缓解超调, 则应该使得 $\alpha (\sum_{k \in \mathbb{K}} h_k + b \nu h_m) \sim D$, 则有: $\alpha = \frac{K}{K + \nu}$ 。

- `'forest'`: 新加入的子树的权重等于丢弃的子树的权重之和。假设被丢弃的子树权重都是 $\frac{1}{K}$, 则此时 $b = 1$, 则有:

$$\alpha \left(\sum_{k \in \mathbb{K}} h_k + b \nu h_m \right) = \alpha \left(\sum_{k \in \mathbb{K}} h_k + \nu h_m \right) \sim \alpha (1 + \nu) D$$

要想缓解超调, 则应该使得 $\alpha (\sum_{k \in \mathbb{K}} h_k + b \nu h_m) \sim D$, 则有: $\alpha = \frac{1}{1 + \nu}$ 。

七、Python API

7.1 数据接口

7.1.1 数据格式

1. `xgboost` 的数据存储在 `DMatrix` 对象中
2. `xgboost` 支持直接从下列格式的文件中加载数据:
 - `libsvm` 文本格式的文件。其格式为:

```
[label] [index1]:[value1] [index2]:[value2] ...
[label] [index1]:[value1] [index2]:[value2] ...
...
```

- `xgboost binary buffer` 文件


```
dtrain = xgb.DMatrix('train.svm.txt') #libsvm 格式
dtest = xgb.DMatrix('test.svm.buffer') # xgboost binary buffer 文件
```

3. `xgboost` 也支持从二维的 `numpy array` 中加载数据

```
data = np.random.rand(5, 10)
label = np.random.randint(2, size=5)
dtrain = xgb.DMatrix(data, label=label)#从 numpy array 中加载
```

4. 你也可以从 `scipy.sparse array` 中加载数据

```
csr = scipy.sparse.csr_matrix((dat, (row, col)))
dtrain = xgb.DMatrix(csr)
```

7.1.2 DMatrix

1. `DMatrix`：由 `xgboost` 内部使用的数据结构，它存储了数据集，并且针对了内存消耗和训练速度进行了优化。

```
xgboost.DMatrix(data, label=None, missing=None, weight=None, silent=False,
                 feature_names=None, feature_types=None, nthread=None)
```

○ 参数：

- `data`：表示数据集。可以为：
 - 一个字符串，表示文件名。数据从该文件中加载
 - 一个二维的 `numpy array`，表示数据集。
- `label`：一个序列，表示样本标记。
- `missing`：一个值，它是缺失值的默认值。
- `weight`：一个序列，给出了数据集中每个样本的权重。
- `silent`：一个布尔值。如果为 `True`，则不输出中间信息。
- `feature_names`：一个字符串序列，给出了每一个特征的名字
- `feature_types`：一个字符串序列，给出了每个特征的数据类型
- `nthread`：

2. 属性：

- `feature_names`：返回每个特征的名字
- `feature_types`：返回每个特征的数据类型

3. 方法：

- `.get_base_margin()`：返回一个浮点数，表示 `DMatrix` 的 `base margin`。
- `.set_base_margin(margin)`：设置 `DMatrix` 的 `base margin`
 - 参数：`margin`：一个序列，给出了每个样本的 `prediction margin`
- `.get_float_info(field)`：返回一个 `numpy array`，表示 `DMatrix` 的 `float property`。

`.set_float_info(field,data)`: 设置 `DMatrix` 的 `float property`。

`.set_float_info_numpy2d(field,data)`: 设置 `DMatrix` 的 `float property`。这里的 `data` 是二维的 `numpy array`

- 参数:

- `field`: 一个字符串, 给出了 `information` 的字段名。注: 意义未知。

- `data`: 一个 `numpy array`, 给出了数据集每一个点的 `float information`

- `.get_uint_info(field)`: 返回 `DMatrix` 的 `unsigned integer property`。

`.set_uint_info(field,data)`: 设置 `DMatrix` 的 `unsigned integer property`。

- 参数:

- `field`: 一个字符串, 给出了 `information` 的字段名。注: 意义未知。

- `data`: 一个 `numpy array`, 给出了数据集每个点的 `uint information`

- 返回值: 一个 `numpy array`, 表示数据集的 `unsigned integer information`

- `.get_label()`: 返回一个 `numpy array`, 表示 `DMatrix` 的 `label`。

`.set_label(label)`: 设置样本标记。

`.set_label_numpy2d(label)`: 设置样本标记。这里的 `label` 为二维的 `numpy array`

- 参数: `label`: 一个序列, 表示样本标记

- `.get_weight()`: 一个 `numpy array`, 返回 `DMatrix` 的样本权重。

`.set_weight(weight)`: 设置样本权重。

`.set_weight_numpy2d(weight)`: 设置样本权重。这里的 `weight` 为二维的 `numpy array`

- 参数: `weight`: 一个序列, 表示样本权重

- `.num_col()`: 返回 `DMatrix` 的列数

- 返回值: 一个整数, 表示特征的数量

- `.num_row()`: 返回 `DMatrix` 的行数

- 返回值: 一个整数, 表示样本的数量

- `save_binary(fname,silent=True)`: 保存 `DMatrix` 到一个 `xgboost buffer` 文件中

- 参数:

- `fname`: 一个字符串, 表示输出的文件名

- `silent`: 一个布尔值。如果为 `True`, 则不输出中间信息。

- `.set_group(group)`: 设置 `DMatrix` 每个组的大小 (用于排序任务)

- 参数: `group`: 一个序列, 给出了每个组的大小

- `slice(rindex)`: 切分 `DMatrix`, 返回一个新的 `DMatrix`。该新的 `DMatrix` 仅仅包含 `rindex`

- 参数: `rindex`: 一个列表, 给出了要保留的 `index`

- 返回值: 一个新的 `DMatrix` 对象

4. 示例:

`data/train.svm.txt` 的内容:

```
1 1:1 2:2
1 1:2 2:3
1 1:3 2:4
1 1:4 2:5
0 1:5 2:6
0 1:6 2:7
0 1:7 2:8
0 1:8 2:9
```

测试代码:

```
import xgboost as xgt
import numpy as np

class MatrixTest:
    ...
    测试 DMatrix
    ...
    def __init__(self):
        self._matrix1 = xgt.DMatrix('data/train.svm.txt')
        self._matrix2 = xgt.DMatrix(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_names=['a', 'b', 'c'],
                                     feature_types=['int', 'int', 'float'], nthread=2)

    def print(self, matrix):
        print('feature_names:%s'%matrix.feature_names)
        print('feature_types:%s' % matrix.feature_types)
    def run_get(self, matrix):
        print('get_base_margin():', matrix.get_base_margin())
        print('get_label():', matrix.get_label())
        print('get_weight():', matrix.get_weight())
        print('num_col():', matrix.num_col())
        print('num_row():', matrix.num_row())

    def test(self):
        print('查看 matrix1 :')
        self.print(self._matrix1)
        # feature_names:['f0', 'f1', 'f2']
        # feature_types:None

        print('\n查看 matrix2 :')
        self.print(self._matrix2)
        # feature_names:['a', 'b', 'c']
        # feature_types:['int', 'int', 'float']

        print('\n查看 matrix1 get:')
        self.run_get(self._matrix1)
        # get_base_margin(): []
```

```

# get_label(): [1. 1. 1. 1. 0. 0. 0. 0.]
# get_weight(): []
# num_col(): 3
# num_row(): 8

print('\n查看 matrix2 get:')
self.run_get(self._matrix2)
# get_base_margin(): []
# get_label(): [1. 2. 3. 4.]
# get_weight(): [0.5 0.4 0.3 0.2]
# num_col(): 3
# num_row(): 4

print(self._matrix2.slice([0,1]).get_label())
# [1. 2.]

```

7.2 模型接口

7.2.1 Booster

1. `Booster` 是 `xgboost` 的模型，它包含了训练、预测、评估等任务的底层实现。

```
xgboost.Booster(params=None, cache=(), model_file=None)
```

参数：

- `params`：一个字典，给出了模型的参数。

该 `Booster` 将调用 `self.set_param(params)` 方法来设置模型的参数。

- `cache`：一个列表，给出了缓存的项。其元素是 `DMatrix` 的对象。模型从这些 `DMatrix` 对象中读取特征名字和特征类型（要求这些 `DMatrix` 对象具有相同的特征名字和特征类型）
- `model_file`：一个字符串，给出了模型文件的位置。

如果给出了 `model_file`，则调用 `self.load_model(model_file)` 来加载模型。

2. 属性：

通过方法来存取、设置属性。

3. 方法：

- `.attr(key)`：获取 `booster` 的属性。如果该属性不存在，则返回 `None`

参数：

- `key`：一个字符串，表示要获取的属性的名字

- `.set_attr(**kwargs)`：设置 `booster` 的属性

- 参数：`kwargs`：关键字参数。注意：参数的值目前只支持字符串。

如果参数的值为 `None`，则表示删除该参数

- `.attributes()`：以字典的形式返回 `booster` 的属性

- `.set_param(params,value=None)`：设置 `booster` 的参数
 - 参数：
 - `params`：一个列表（元素为键值对）、一个字典、或者一个字符串。表示待设置的参数
 - `value`：如果 `params` 为字符串，那么 `params` 就是键，而 `value` 就是参数值。
- `.boost(dtrain,grad,hess)`：执行一次训练迭代
 - 参数：
 - `dtrain`：一个 `DMatrix` 对象，表示训练集
 - `grad`：一个列表，表示一阶的梯度
 - `hess`：一个列表，表示二阶的偏导数
- `.update(dtrain,iteration,fobj=None)`：对一次迭代进行更新
 - 参数：
 - `dtrain`：一个 `DMatrix` 对象，表示训练集
 - `iteration`：一个整数，表示当前的迭代步数编号
 - `fobj`：一个函数，表示自定义的目标函数

由于 `Booster` 没有 `.train()` 方法，因此需要用下面的策略进行迭代：

```
for i in range(0,100):
    booster.update(train_matrix,iteration=i)
```

- `.copy()`：拷贝当前的 `booster`，并返回一个新的 `Booster` 对象
- `.dump_model(fout,fmap='',with_stats=False)`：dump 模型到一个文本文件中。
 - 参数：
 - `fout`：一个字符串，表示输出文件的文件名
 - `fmap`：一个字符串，表示存储 `feature map` 的文件的文件名。`booster` 需要从它里面读取特征的信息。
该文件每一行依次代表一个特征。每一行的格式为：`feature name:feature type`。其中 `feature type` 为 `int`、`float` 等表示数据类型的字符串。
 - `with_stats`：一个布尔值。如果为 `True`，则输出 `split` 的统计信息
- `.get_dump(fmap='',with_stats=False,dump_format='text')`：dump 模型为字符的列表（而不是存到文件中）。
 - 参数：
 - `fmap`：一个字符串，表示存储 `feature map` 的文件的文件名。`booster` 需要从它里面读取特征的信息。
 - `with_stats`：一个布尔值。如果为 `True`，则输出 `split` 的统计信息
 - `dump_format`：一个字符串，给出了输出的格式
 - 返回值：一个字符串的列表。每个字符串描述了一棵子树。
- `.eval(data,name='eval',iteration=0)`：对模型进行评估
 - 参数：
 - `data`：一个 `DMatrix` 对象，表示数据集
 - `name`：一个字符串，表示数据集的名字

- `iteration`：一个整数，表示当前的迭代编号
 - 返回值：一个字符串，表示评估结果
- `.eval_set(evals, iteration=0, feval=None)`：评估一系列的数据集
 - 参数：
 - `evals`：一个列表，列表元素为元组 (`DMatrix, string`)，它给出了待评估的数据集
 - `iteration`：一个整数，表示当前的迭代编号
 - `feval`：一个函数，给出了自定义的评估函数
 - 返回值：一个字符串，表示评估结果
- `.get_fscore(fmap='')`：返回每个特征的重要性
 - 参数：
 - `fmap`：一个字符串，给出了 `feature map` 文件的文件名。`booster` 需要从它里面读取特征的信息。
 - 返回值：一个字典，给出了每个特征的重要性
- `.get_score(fmap='', importance_type='weight')`：返回每个特征的重要性
 - 参数：
 - `fmap`：一个字符串，给出了 `feature map` 文件的文件名。`booster` 需要从它里面读取特征的信息。
 - `importance_type`：一个字符串，给出了特征的衡量指标。可以为：
 - `'weight'`：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：该特征在树的 `'cover'` 中，获取的平均增益。
 - 返回值：一个字典，给出了每个特征的重要性
- `.get_split_value_histogram(feature, fmap='', bins=None, as_pandas=True)`：获取一个特征的划分 `value histogram`。
 - 参数：
 - `feature`：一个字符串，给出了划分特征的名字
 - `fmap`：一个字符串，给出了 `feature map` 文件的文件名。`booster` 需要从它里面读取特征的信息。
 - `bins`：最大的分桶的数量。如果 `bins=None` 或者 `bins>n_unique`，则分桶的数量实际上等于 `n_unique`。其中 `n_unique` 是划分点的值的 `unique`
 - `as_pandas`：一个布尔值。如果为 `True`，则返回一个 `pandas.DataFrame`；否则返回一个 `numpy ndarray`。
 - 返回值：以一个 `numpy ndarray` 或者 `pandas.DataFrame` 形式返回的、代表拆分点的 `histogram` 的结果。
- `.load_model(fname)`：从文件中加载模型。
 - 参数： `fname`：一个文件或者一个内存 `buffer`，`xgboost` 从它加载模型
- `.save_model(fname)`：保存模型到文件中
 - 参数： `fname`：一个字符串，表示文件名
- `save_raw()`：将模型保存成内存 `buffer`
 - 返回值：一个内存 `buffer`，代表该模型
- `.load_rabit_checkpoint()`：从 `rabit checkpoint` 中初始化模型。

- 返回值：一个整数，表示模型的版本号

- `.predict(data,output_margin=False,ntree_limit=0,pred_leaf=False,pred_contribs=False,approx_contribs=False)`：执行预测

该方法不是线程安全的。对于每个 `booster` 来讲，你只能在某个线程中调用它的 `.predict` 方法。如果你在多个线程中调用 `.predict` 方法，则可能会有问题。

要想解决该问题，你必须在每个线程中调用 `booster.copy()` 来拷贝该 `booster` 到每个线程中

- 参数：

- `data`：一个 `DMatrix` 对象，表示测试集
- `output_margin`：一个布尔值。表示是否输出原始的、未经过转换的 `margin value`
- `ntree_limit`：一个整数。表示使用多少棵子树来预测。默认值为0，表示使用所有的子树。

如果训练的时候发生了早停，则你可以使用 `booster.best_ntree_limit`。

- `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。

每个子树的叶节点都是从 1 开始编号的。

- `pred_contribs`：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。

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。

- `approx_contribs`：一个布尔值。如果为 `True`，则大致估算出每个特征的贡献程度。

- 返回值：一个 `ndarray`，表示预测结果

4. `Booster` 没有 `train` 方法。因此有两种策略来获得训练好的 `Booster`

- 从训练好的模型的文件中 `.load_model()` 来获取
- 多次调用 `.update()` 方法

5. 示例：

```
import xgboost as xgb
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

_label_map={
    # 'Iris-setosa':0, #经过裁剪的，去掉了 iris 中的 setosa 类
    'Iris-versicolor':0,
    'Iris-virginica':1
}

class BoosterTest:
    ...

    测试 Booster
    ...

    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)
self._train_matrix=xgt.DMatrix(data=train_X,label=train_Y,
        eature_names=_feature_names,
        feature_types=['float','float','float','float'])
self._validate_matrix = xgt.DMatrix(data=test_X, label=test_Y,
        feature_names=_feature_names,
        feature_types=['float','float','float','float'])
self._booster=xgt.Booster(params={
    'booster':'gbtree',
    'silent':0,#打印消息
    'eta':0.1, #学习率
    'max_depth':5,
    'tree_method':'exact',
    'objective':'binary:logistic',
    'eval_metric':'auc',
    'seed':321},
    cache=[self._train_matrix,self._validate_matrix])

def test_attribute(self):
    """
    测试属性的设置和获取
    :return:
    """
    self._booster.set_attr(key1= '1')
    print('attr:key1 -> ',self._booster.attr('key1'))
    print('attr:key2 -> ',self._booster.attr('key2'))
    print('attributes -> ',self._booster.attributes())
def test_dump_model(self):
    """
    测试 dump 模型
    :return:
    """
    _dump_str=self._booster.get_dump(fmap='model/booster.feature',
        with_stats=True,dump_format='text')
    print('dump:',_dump_str[0][:20]+'...' if _dump_str else [])
    self._booster.dump_model('model/booster.model',
        fmap='model/booster.feature',with_stats=True)
def test_train(self):
    """
    训练
    :return:
    """
    for i in range(0,100):
        self._booster.update(self._train_matrix,iteration=i)
        print(self._booster.eval(self._train_matrix, name='train', iteration=i))
        print(self._booster.eval(self._validate_matrix,name='eval',iteration=i))

```

```

def test_importance(self):
    """
    测试特征重要性
    :return:
    """

    print('fscore:', self._booster.get_fscore('model/booster.feature'))
    print('score.weight:', self._booster.get_score(importance_type='weight'))
    print('score.gain:', self._booster.get_score(importance_type='gain'))

def test(self):
    self.test_attribute()
    # attr:key1 -> 1
    # attr:key2 -> None
    # attributes -> {'key1': '1'}
    self.test_dump_model()
    # dump: []
    self.test_train()
    # [0] train-auc:0.980816
    # [0] eval-auc:0.933333
    # ...
    # [99] train-auc:0.998367
    # [99] eval-auc:0.995556
    self.test_dump_model()
    # dump: 0:[f2<4.85] yes=1,no...
    self.test_importance()
    # score: {'f2': 80, 'f3': 72, 'f0': 6, 'f1': 5}
    # score.weight: {'Petal Length': 80, 'Petal Width': 72, 'Sepal Length': 6, 'Sepal
    Width': 5}
    # score.gain: {'Petal Length': 3.6525380337500004, 'Petal Width': 2.2072901486111114,
    'Sepal Length': 0.06247816666666667, 'Sepal Width': 0.09243024}

if __name__ == '__main__':
    BoosterTest().test()

```

7.2.2 直接学习

1. `xgboost.train()`: 使用给定的参数来训练一个 `booster`

```

xgboost.train(params, dtrain, num_boost_round=10, evals=(), obj=None, feval=None,
               maximize=False, early_stopping_rounds=None, evals_result=None, verbose_eval=True,
               xgb_model=None, callbacks=None, learning_rates=None)

```

o 参数:

- `params`: 一个列表 (元素为键值对)、一个字典, 表示训练的参数
- `dtrain`: 一个 `DMatrix` 对象, 表示训练集
- `num_boost_round`: 一个整数, 表示 `boosting` 迭代数量
- `evals`: 一个列表, 元素为 `(DMatrix, string)`。它给出了训练期间的验证集, 以及验证集的名字 (从而区分验证集的评估结果)。

- `obj`: 一个函数, 它表示自定义的目标函数
- `feval`: 一个函数, 它表示自定义的 `evaluation` 函数
- `maximize`: 一个布尔值。如果为 `True`, 则表示是对 `feval` 求最大值; 否则为求最小值
- `early_stopping_rounds`: 一个整数, 表示早停参数。

如果在 `early_stopping_rounds` 个迭代步内, 验证集的验证误差没有下降, 则训练停止。

- 该参数要求 `evals` 参数至少包含一个验证集。如果 `evals` 参数包含了多个验证集, 则使用最后的一个。
 - 返回的模型是最后一次迭代的模型 (而不是最佳的模型)。
 - 如果早停发生, 则模型拥有三个额外的字段:
 - `.best_score`: 最佳的分数
 - `.best_iteration`: 最佳的迭代步数
 - `.best_ntree_limit`: 最佳的子模型数量
 - `evals_result`: 一个字典, 它给出了对测试集要进行评估的指标。
 - `verbose_eval`: 一个布尔值或者整数。
 - 如果为 `True`, 则 `evaluation metric` 将在每个 `boosting stage` 打印出来
 - 如果为一个整数, 则 `evaluation metric` 将在每隔 `verbose_eval` 个 `boosting stage` 打印出来。另外最后一个 `boosting stage`, 以及早停的 `boosting stage` 的 `evaluation metric` 也会被打印
 - `learning_rates`: 一个列表, 给出了每个迭代步的学习率。
 - 你可以让学习率进行衰减
 - `xgb_model`: 一个 `Booster` 实例, 或者一个存储了 `xgboost` 模型的文件的文件名。它给出了待训练的模型。
- 这种做法允许连续训练。
- `callbacks`: 一个回调函数的列表, 它给出了在每个迭代步结束之后需要调用的那些函数。
- 你可以使用 `xgboost` 中预定义的一些回调函数 (位于 `xgboost.callback` 模块) 。如:
- ```
xgboost.reset_learning_rate(custom_rates)
```

◦ 返回值: 一个 `Booster` 对象, 表示训练好的模型

2. `xgboost.cv()`: 使用给定的参数执行交叉验证。它常用作参数搜索。

```
xgboost.cv(params, dtrain, num_boost_round=10, nfold=3, stratified=False, folds=None,
 metrics=(), obj=None, feval=None, maximize=False, early_stopping_rounds=None,
 fpreproc=None, as_pandas=True, verbose_eval=None, show_stdv=True, seed=0,
 callbacks=None, shuffle=True)
```

◦ 参数:

- `params`: 一个列表 (元素为键值对)、一个字典, 表示训练的参数
- `dtrain`: 一个 `DMatrix` 对象, 表示训练集
- `num_boost_round`: 一个整数, 表示 `boosting` 迭代数量
- `nfold`: 一个整数, 表示交叉验证的 `fold` 的数量

- `stratified`：一个布尔值。如果为 `True`，则执行分层采样
  - `folds`：一个 `scikit-learn` 的 `KFold` 实例或者 `StratifiedKFold` 实例。
  - `metrics`：一个字符串或者一个字符串的列表，指定了交叉验证时的 `evaluation metrics`  
如果同时在 `params` 里指定了 `eval_metric`，则 `metrics` 参数优先。
  - `obj`：一个函数，它表示自定义的目标函数
  - `feval`：一个函数，它表示自定义的 `evaluation` 函数
  - `maximize`：一个布尔值。如果为 `True`，则表示是对 `feval` 求最大值；否则为求最小值
  - `early_stopping_rounds`：一个整数，表示早停参数。  
如果在 `early_stopping_rounds` 个迭代步内，验证集的验证误差没有下降，则训练停止。
    - 返回 `evaluation history` 结果中的最后一项是最佳的迭代步的评估结果
  - `fpreproc`：一个函数。它是预处理函数，其参数为 `(dtrain, dtest, param)`，返回值是经过了变换之后的 `(dtrain, dtest, param)`
  - `as_pandas`：一个布尔值。如果为 `True`，则返回一个 `pandas.DataFrame`；否则返回一个 `numpy.ndarray`
  - `verbose_eval`：参考 `xgboost.train()`
  - `show_stdv`：一个布尔值。是否 `verbose` 中打印标准差。  
它对返回结果没有影响。返回结果始终包含标准差。
  - `seed`：一个整数，表示随机数种子
  - `callbacks`：参考 `xgboost.train()`
  - `shuffle`：一个布尔值。如果为 `True`，则创建 `folds` 之前先混洗数据。
- 返回值：一个字符串的列表，给出了 `evaluation history`。它给的是早停时刻的 `history` (此时对应着最优模型)，早停之后的结果被抛弃。

### 3. 示例：

```
class TrainTest:
 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)
 self._train_matrix = xgt.DMatrix(data=train_X, label=train_Y,
 feature_names=_feature_names,
 feature_types=['float', 'float', 'float', 'float'])
 self._validate_matrix = xgt.DMatrix(data=test_X, label=test_Y,
 feature_names=_feature_names,
 feature_types=['float', 'float', 'float', 'float'])

 def train_test(self):
 params={
```

```

 'booster': 'gbtree',
 'eta': 0.01,
 'max_depth': 5,
 'tree_method': 'exact',
 'objective': 'binary:logistic',
 'eval_metric': ['logloss', 'error', 'auc']
 }
 eval_rst={}
 booster=xgt.train(params,self._train_matrix,num_boost_round=20,
 evals=([(self._train_matrix, 'valid1'),(self._validate_matrix, 'valid2')]),
 early_stopping_rounds=5,evals_result=eval_rst,verbose_eval=True)
 ## 训练输出
 # Multiple eval metrics have been passed: 'valid2-auc' will be used for early
 stopping.
 # Will train until valid2-auc hasn't improved in 5 rounds.
 # [0] valid1-logloss:0.685684 valid1-error:0.042857 valid1-auc:0.980816 valid2-
 logloss:0.685749 valid2-error:0.066667 valid2-auc:0.933333
 # ...
 # Stopping. Best iteration:
 # [1] valid1-logloss:0.678149 valid1-error:0.042857 valid1-auc:0.99551 valid2-
 logloss:0.677882 valid2-error:0.066667 valid2-auc:0.966667

 print('booster attributes:',booster.attributes())
 # booster attributes: {'best_iteration': '1', 'best_msg': '[1]\tvalid1-
 logloss:0.678149\tvalid1-error:0.042857\tvalid1-auc:0.99551\tvalid2-
 logloss:0.677882\tvalid2-error:0.066667\tvalid2-auc:0.966667', 'best_score': '0.966667'}

 print('fscore:', booster.get_fscore())
 # fscore: {'Petal Length': 8, 'Petal Width': 7}

 print('eval_rst:',eval_rst)
 # eval_rst: {'valid1': {'logloss': [0.685684, 0.678149, 0.671075, 0.663787, 0.656948,
 0.649895], 'error': [0.042857, 0.042857, 0.042857, 0.042857, 0.042857, 0.042857], 'auc':
 [0.980816, 0.99551, 0.99551, 0.99551, 0.99551, 0.99551]}, 'valid2': {'logloss':
 [0.685749, 0.677882, 0.670747, 0.663147, 0.656263, 0.648916], 'error': [0.066667,
 0.066667, 0.066667, 0.066667, 0.066667], 'auc': [0.933333, 0.966667, 0.966667,
 0.966667, 0.966667, 0.966667]}}

 def cv_test(self):
 params = {
 'booster': 'gbtree',
 'eta': 0.01,
 'max_depth': 5,
 'tree_method': 'exact',
 'objective': 'binary:logistic',
 'eval_metric': ['logloss', 'error', 'auc']
 }

 eval_history = xgt.cv(params, self._train_matrix,num_boost_round=20,
 nfold=3,stratified=True,metrics=['error', 'auc'],
 early_stopping_rounds=5,verbose_eval=True,shuffle=True)

```

```

训练输出
[0] train-auc:0.974306+0.00309697 train-error:0.0428743+0.0177703 test-
auc:0.887626+0.0695933 test-error:0.112374+0.0695933
#....
print('eval_history:', eval_history)
eval_history: test-auc-mean test-auc-std test-error-mean test-error-std \
0 0.887626 0.069593 0.112374 0.069593
1 0.925821 0.020752 0.112374 0.069593
2 0.925821 0.020752 0.098485 0.050631

train-auc-mean train-auc-std train-error-mean train-error-std
0 0.974306 0.003097 0.042874 0.01777
1 0.987893 0.012337 0.042874 0.01777
2 0.986735 0.011871 0.042874 0.01777

```

## 7.2.3 Scikit-Learn API

1. `xgboost` 给出了针对 `scikit-learn` 接口的 API
2. `xgboost.XGBRegressor`：它实现了 `scikit-learn` 的回归模型 API

```

class xgboost.XGBRegressor(max_depth=3, learning_rate=0.1, n_estimators=100,
 silent=True, objective='reg:linear', booster='gbtree', n_jobs=1, nthread=None,
 gamma=0, min_child_weight=1, max_delta_step=0, subsample=1, colsample_bytree=1,
 colsample_bylevel=1, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
 base_score=0.5, random_state=0, seed=None, missing=None, **kwargs)

```

参数：

- `max_depth`：一个整数，表示子树的最大深度
- `learning_rate`：一个浮点数，表示学习率
- `n_estimators`：一个整数，表示预期需要学习的子树的数量
- `silent`：一个布尔值。如果为 `False`，则打印中间信息
- `objective`：一个字符串或者可调用对象，指定了目标函数。其函数签名为：  
`objective(y_true, y_pred) -> grad, hess`。其中：
  - `y_true`：一个形状为 `[n_sample]` 的序列，表示真实的标签值
  - `y_pred`：一个形状为 `[n_sample]` 的序列，表示预测的标签值
  - `grad`：一个形状为 `[n_sample]` 的序列，表示每个样本处的梯度
  - `hess`：一个形状为 `[n_sample]` 的序列，表示每个样本处的二阶偏导数
- `booster`：一个字符串。指定了用哪一种基模型。可以为：`'gbtree'`, `'gblinear'`, `'dart'`
- `n_jobs`：一个整数，指定了并行度，即开启多少个线程来训练。如果为 `-1`，则使用所有的 CPU
- `gamma`：一个浮点数，也称作最小划分损失 `min_split_loss`。它刻画的是：对于一个叶子节点，当对它采取划分之后，损失函数的降低值的阈值。
  - 如果大于该阈值，则该叶子节点值得继续划分
  - 如果小于该阈值，则该叶子节点不值得继续划分

- `min_child_weight`: 一个整数, 子节点的权重阈值。它刻画的是: 对于一个叶子节点, 当对它采取划分之后, 它的所有子节点的权重之和的阈值。
  - 如果它的所有子节点的权重之和大于该阈值, 则该叶子节点值得继续划分
  - 如果它的所有子节点的权重之和小于该阈值, 则该叶子节点不值得继续划分

所谓的权重:

- 对于线性模型(`booster=gblinear`), 权重就是: 叶子节点包含的样本数量。因此该参数就是每个节点包含的最少样本数量。
- 对于树模型(`booster=gbtrees, dart`), 权重就是: 叶子节点包含样本的所有二阶偏导数之和。
- `max_delta_step`: 一个整数, 每棵树的权重估计时的最大 `delta_step`。取值范围为  $[0, \infty]$ , 0 表示没有限制, 默认值为 0。
- `subsample`: 一个浮点数, 对训练样本的采样比例。取值范围为  $(0, 1]$ , 默认值为 1。  
如果为 `0.5`, 表示随机使用一半的训练样本来训练子树。它有助于缓解过拟合。
- `colsample_bytree`: 一个浮点数, 构建子树时, 对特征的采样比例。取值范围为  $(0, 1]$ , 默认值为 1。  
如果为 `0.5`, 表示随机使用一半的特征来训练子树。它有助于缓解过拟合。
- `colsample_bylevel`: 一个浮点数, 寻找划分点时, 对特征的采样比例。取值范围为  $(0, 1]$ , 默认值为 1。  
如果为 `0.5`, 表示随机使用一半的特征来寻找最佳划分点。它有助于缓解过拟合。
- `reg_alpha`: 一个浮点数, 是 L1 正则化系数。它是 `xgb` 的 `alpha` 参数
- `reg_lambda`: 一个浮点数, 是 L2 正则化系数。它是 `xgb` 的 `lambda` 参数
- `scale_pos_weight`: 一个浮点数, 用于调整正负样本的权重, 常用于类别不平衡的分类问题。默认为 1。

一个典型的参数值为: `负样本数量/正样本数量`

- `base_score`: 一个浮点数, 给所有样本的一个初始的预测得分。它引入了全局的 `bias`
  - `random_state`: 一个整数, 表示随机数种子。
  - `missing`: 一个浮点数, 它的值代表发生了数据缺失。默认为 `np.nan`
  - `kwargs`: 一个字典, 给出了关键字参数。它用于设置 `Booster` 对象
3. `xgboost.XGBClassifier`: 它实现了 `scikit-learn` 的分类模型 API

```
class xgboost.XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100,
 silent=True, objective='binary:logistic', booster='gbtree', n_jobs=1,
 nthread=None, gamma=0, min_child_weight=1, max_delta_step=0, subsample=1,
 colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1,
 scale_pos_weight=1, base_score=0.5, random_state=0, seed=None,
 missing=None, **kwargs)
```

参数参考 `xgboost.XGBRegressor`

4. `xgboost.XGBClassifier` 和 `xgboost.XGBRegressor` 的方法:
- `.fit()`: 训练模型



```
fit(X, y, sample_weight=None, eval_set=None, eval_metric=None,
 early_stopping_rounds=None, verbose=True, xgb_model=None)
```

参数:

- `X`: 一个 `array-like`, 表示训练集
- `y`: 一个序列, 表示标记
- `sample_weight`: 一个序列, 给出了每个样本的权重
- `eval_set`: 一个列表, 元素为 `(X,y)`, 给出了验证集及其标签。它们用于早停。

如果有多个验证集, 则使用最后一个

- `eval_metric`: 一个字符串或者可调对象, 用于 `evaluation metric`
  - 如果为字符串, 则是内置的度量函数的名字
  - 如果为可调对象, 则它的签名为 `(y_pred,y_true)==>(str,value)`
- `early_stopping_rounds`: 指定早停的次数。参考 `xgboost.train()`
- `verbose`: 一个布尔值。如果为 `True`, 则打印验证集的评估结果。
- `xgb_model`: 一个 `Booster` 实例, 或者一个存储了 `xgboost` 模型的文件的文件名。它给出了待训练的模型。

这种做法允许连续训练。

- `.predict()`: 执行预测

```
predict(data, output_margin=False, ntree_limit=0)
```

参数:

- `data`: 一个 `DMatrix` 对象, 表示测试集
- `output_margin`: 一个布尔值。表示是否输出原始的、未经过转换的 `margin value`
- `ntree_limit`: 一个整数。表示使用多少棵子树来预测。默认值为0, 表示使用所有的子树。

如果训练的时候发生了早停, 则你可以使用 `booster.best_ntree_limit`。

返回值: 一个 `ndarray`, 表示预测结果

- 对于回归问题, 返回的就是原始的预测结果
- 对于分类问题, 返回的就是预测类别(阈值为 0.5)

- `.predict_proba(data, output_margin=False, ntree_limit=0)`: 执行预测, 预测的是各类别的概率

参数: 参考 `.predict()`

返回值: 一个 `ndarray`, 表示预测结果

它只用于分类问题, 返回的是预测各类别的概率

- `.evals_result()`: 返回一个字典, 给出了各个验证集在各个验证参数上的历史值

它不同于 `cv()` 函数的返回值。`cv()` 函数返回 `evaluation history` 是早停时刻的。而这里返回的是所有的历史值

5. 示例:

```

class SKLTest:
 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])

 self.train_X, self.test_X, self.train_Y, self.test_Y = \
 train_test_split(x, y, test_size=0.3, stratify=y, shuffle=True, random_state=1)

 def train_test(self):
 clf=xgt.XGBClassifier(max_depth=3,learning_rate=0.1,n_estimators=100)
 clf.fit(self.train_X,self.train_Y,eval_metric='auc',
 eval_set=[(self.test_X,self.test_Y)],
 early_stopping_rounds=3)
 # 训练输出:
 # Will train until validation_0-auc hasn't improved in 3 rounds.
 # [0] validation_0-auc:0.933333
 # ...
 # Stopping. Best iteration:
 # [2] validation_0-auc:0.997778
 print('evals_result:',clf.evals_result())
 # evals_result: {'validation_0': {'auc': [0.933333, 0.966667, 0.997778, 0.997778,
 0.997778]}}
 print('predict:',clf.predict(self.test_X))
 # predict: [1 1 0 0 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 0 0 1 1 0 0 1 1 0]

```

## 7.3 绘图API

1. `xgboost.plot_importance()`：绘制特征重要性

```

xgboost.plot_importance(booster, ax=None, height=0.2, xlim=None, ylim=None,
 title='Feature importance', xlabel='F score', ylabel='Features',
 importance_type='weight', max_num_features=None, grid=True,
 show_values=True, **kwargs)

```

参数：

- `booster`：一个 `Booster` 对象，一个 `XGBModel` 对象，或者由 `Booster.get_fscore()` 返回的字典
- `ax`：一个 `matplotlib Axes` 对象。特征重要性将绘制在它上面。  
如果为 `None`，则新建一个 `Axes`
- `grid`：一个布尔值。如果为 `True`，则开启 `axes grid`
- `importance_type`：一个字符串，指定了特征重要性的类别。参考 `Booster.get_fscore()`
- `max_num_features`：一个整数，指定展示的特征的最大数量。如果为 `None`，则展示所有的特征
- `height`：一个浮点数，指定 `bar` 的高度。它传递给 `ax.barh()`
- `xlim`：一个元组，传递给 `axes.xlim()`

- `ylim`: 一个元组, 传递给 `axes.ylim()`
- `title`: 一个字符串, 设置 `Axes` 的标题。默认为 "Feature importance"。如果为 `None`, 则没有标题
- `xlabel`: 一个字符串, 设置 `Axes` 的 X 轴标题。默认为 "F score"。如果为 `None`, 则 X 轴没有标题
- `ylabel`: 一个字符串, 设置 `Axes` 的 Y 轴标题。默认为 "Features"。如果为 `None`, 则 Y 轴没有标题
- `show_values`: 一个布尔值。如果为 `True`, 则在绘图上展示具体的值。
- `kwargs`: 关键字参数, 用于传递给 `ax.barh()`

返回 `ax` (一个 `matplotlib Axes` 对象)

## 2. `xgboost.plot_tree()`: 绘制指定的子树。

```
xgboost.plot_tree(booster, fmap='', num_trees=0, rankdir='UT', ax=None, **kwargs)
```

参数:

- `booster`: 一个 `Booster` 对象, 一个 `XGBModel` 对象
- `fmap`: 一个字符串, 给出了 `feature map` 文件的文件名
- `num_trees`: 一个整数, 制定了要绘制的子数的编号。默认为 0
- `rankdir`: 一个字符串, 它传递给 `graphviz` 的 `graph_attr`
- `ax`: 一个 `matplotlib Axes` 对象。特征重要性将绘制在它上面。  
如果为 `None`, 则新建一个 `Axes`
- `kwargs`: 关键字参数, 用于传递给 `graphviz` 的 `graph_attr`

返回 `ax` (一个 `matplotlib Axes` 对象)

## 3. `xgboost.tp_graphviz()`: 转换指定的子树成一个 `graphviz` 实例。

在 `IPython` 中, 可以自动绘制 `graphviz` 实例; 否则你需要手动调用 `graphviz` 对象的 `.render()` 方法来绘制。

```
xgboost.to_graphviz(booster, fmap='', num_trees=0, rankdir='UT', yes_color='#0000FF',
 no_color='#FF0000', **kwargs)
```

参数:

- `yes_color`: 一个字符串, 给出了满足 `node condition` 的边的颜色
- `no_color`: 一个字符串, 给出了不满足 `node condition` 的边的颜色
- 其它参数参考 `xgboost.plot_tree()`

返回 `ax` (一个 `matplotlib Axes` 对象)

## 4. 示例:

```
class PlotTest:
 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)
self._train_matrix = xgt.DMatrix(data=train_X, label=train_Y,
 feature_names=_feature_names,
 feature_types=['float', 'float', 'float', 'float'])
self._validate_matrix = xgt.DMatrix(data=test_X, label=test_Y,
 feature_names=_feature_names,
 feature_types=['float', 'float', 'float', 'float'])

def plot_test(self):
 params = {
 'booster': 'gbtree',
 'eta': 0.01,
 'max_depth': 5,
 'tree_method': 'exact',
 'objective': 'binary:logistic',
 'eval_metric': ['logloss', 'error', 'auc']
 }
 eval_rst = {}
 booster = xgt.train(params, self._train_matrix,
 num_boost_round=20, evals=([self._train_matrix, 'valid1'),
 (self._validate_matrix, 'valid2')]),
 early_stopping_rounds=5, evals_result=eval_rst, verbose_eval=True)
 xgt.plot_importance(booster)
 plt.show()

```

