LightGBM算法原理小结

原创 石头 机器学习算法那些事 2019-02-14

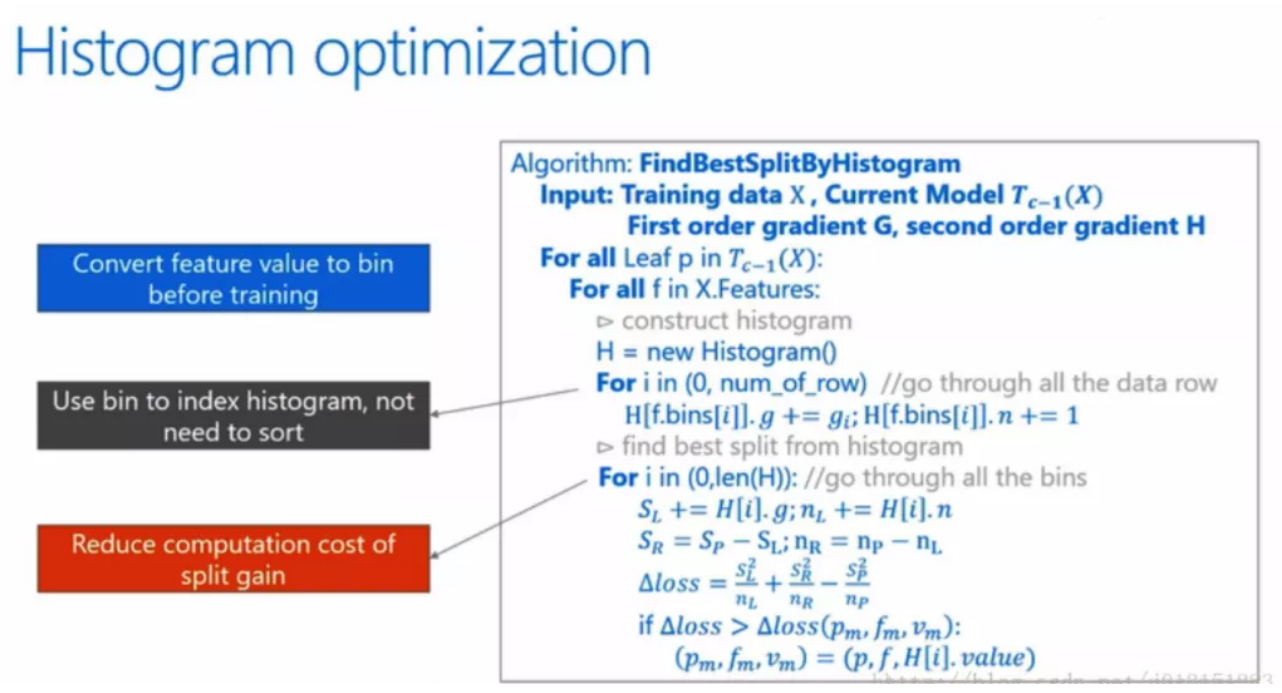
提升树是利用加法模型和前向分布算法实现学习的优化过程，它有一些高效的实现，如GBDT，XGBoost和pGBRT，其中GBDT是通过损失函数的负梯度拟合残差，XGBoost则是利用损失函数的二阶导展开式拟合残差。但是，当面对大量数据集和高维特征时，其扩展性和效率很难令人满意，最主要的原因是对于每一个特征，它们需要扫描所有的样本数据来获得最优切分点，这个过程是非常耗时的 。本文介绍基于GBDT的另一种形式LightGBM，LightGBM是基于直方图的切分点算法，其很好的解决了这些问题 ，本文对LightGBM的算法原理进行了总结。

目录

1. 基于直方图的切分点算法
2. 直方图算法改进
3. 直方图做差优化
4. 树的生长策略
5. 支持类别特征
6. 支持高效并行
7. 与XGBoost的对比

1. 基于直方图的切分点算法

算法流程图如下：



流程解释：LightGBM通过直方图算法把连续的特征值离散化成对应的bin（可以理解成桶），然后累加每个bin对应特征的梯度值并计数，最后遍历所有特征和数据，寻找最优切分点 。下面详细解释这一过程 。

1） 直方图算法：

直方图通过分段函数把连续值离散化成对应的bin。

如分段函数为：

$$f(x)=\begin{cases}0 & x\in[0,5)\\ 1 & x\in[5,10)\\ 2 & x\in[10,15)\\ 3 & x\in[15,\infty)\end{cases}$$

那么，对于连续值≥0的特征分成四个桶，特征的所有可能切分点个数降为4，即bin的个数 ，因此大大加快了训练速度。

直方图每个bin包含了两类信息，分别是每个bin样本中的梯度之和与每个bin中的样本数量，对应于流程图的表达式：

$$H[f.bins[i]].g+=g_i$$
$$H[f.bins[i]].n+=1$$

每个bin累加的梯度包含了 一阶梯度与二阶梯度 ，如下代码（LightGBM/src/io/sparse_bin.hpp）：

```
void ConstructHistogram(int leaf, const score_t* gradient, const score_t* hessian,
                        HistogramBinEntry* out) const override {
    // get current leaf boundary
    const data_size_t start = leaf_start_[leaf];
    const data_size_t end = start + leaf_cnt_[leaf];
    const int rest = (end - start) % 4;
    data_size_t i = start;
    // use data on current leaf to construct histogram
    for (; i < end - rest; i += 4) {
      const VAL_T bin0 = ordered_pair_[i].bin;
      const VAL_T bin1 = ordered_pair_[i + 1].bin;
      const VAL_T bin2 = ordered_pair_[i + 2].bin;
      const VAL_T bin3 = ordered_pair_[i + 3].bin;

      const auto g0 = gradient[ordered_pair_[i].ridx];
      const auto h0 = hessian[ordered_pair_[i].ridx];
      const auto g1 = gradient[ordered_pair_[i + 1].ridx];
      const auto h1 = hessian[ordered_pair_[i + 1].ridx];
      const auto g2 = gradient[ordered_pair_[i + 2].ridx];
      const auto h2 = hessian[ordered_pair_[i + 2].ridx];
      const auto g3 = gradient[ordered_pair_[i + 3].ridx];
      const auto h3 = hessian[ordered_pair_[i + 3].ridx];

      out[bin0].sum_gradients += g0;        // 累加一阶梯度
      out[bin1].sum_gradients += g1;
      out[bin2].sum_gradients += g2;
      out[bin3].sum_gradients += g3;
```

```
      out[bin0].sum_hessians += h0;        // 累加二阶梯度
      out[bin1].sum_hessians += h1;
      out[bin2].sum_hessians += h2;
      out[bin3].sum_hessians += h3;

      ++out[bin0].cnt;                     // 累加个数
      ++out[bin1].cnt;
      ++out[bin2].cnt;
      ++out[bin3].cnt;
    }

    for (; i < end; ++i) {
      const VAL_T bin0 = ordered_pair_[i].bin;

      const auto g0 = gradient[ordered_pair_[i].ridx];
      const auto h0 = hessian[ordered_pair_[i].ridx];

      out[bin0].sum_gradients += g0;
      out[bin0].sum_hessians += h0;
      ++out[bin0].cnt;
    }
  }
```

2）切分点算法：

LightGBM和XGBoost的切分点算法思想是一样的，比较切分后的增益与设定的最小增益的大小，若大于，则切分；反之，则不切分该节点。下面详细叙述这一过程（该节代码实现均在LightGBM/src/treelearner/feature_histogram.hpp）：

（1）计算每个叶子节点的输出wj；

```
static double CalculateSplittedLeafOutput(double sum_gradients, double sum_hessians, double l1,
    double ret = -ThresholdL1(sum_gradients, l1) / (sum_hessians + l2);    // 计算叶子节点输出
    if (max_delta_step <= 0.0f || std::fabs(ret) <= max_delta_step) {
      return ret;
    } else {
      return Common::Sign(ret) * max_delta_step;
    }
  }
```

其中，thresholdL1函数含义为如下代码：

```
static double ThresholdL1(double s, double l1) {
    const double reg_s = std::max(0.0, std::fabs(s) - l1);    // 比较s与l1的大小
    return Common::Sign(s) * reg_s;
  }
```

根据上面两图的代码，可知lightGBM的叶子节点输出与XGBoost类似，即：

$$w_j = -\frac{G_j}{H_j + \lambda}$$

其中，Gj为叶子节点包含所有样本的一阶梯度之和，Hj为二阶梯度之和 。

（2）计算节点切分后的分数

```
static double GetSplitGains(double sum_left_gradients, double sum_left_hessians,
                            double sum_right_gradients, double sum_right_hessians,
                            double l1, double l2, double max_delta_step,
                            double min_constraint, double max_constraint, int8_t monotone_cons
      // 计算左叶子节点的输出，
      double left_output = CalculateSplittedLeafOutput(sum_left_gradients, sum_left_hessians,
      // 计算右叶子节点的输出
      double right_output = CalculateSplittedLeafOutput(sum_right_gradients, sum_right_hessian
    if (((monotone_constraint > 0) && (left_output > right_output)) ||
      ((monotone_constraint < 0) && (left_output < right_output))) {
      return 0;
    }
      // 节点切分后的分数等于左叶子节点分数与右叶子节点分数之和
    return GetLeafSplitGainGivenOutput(sum_left_gradients, sum_left_hessians, l1, l2, left_outpu
      + GetLeafSplitGainGivenOutput(sum_right_gradients, sum_right_hessians, l1, l2, right_outpu
  }
```

叶子节点分数为GetLeafSplitGainGivenOutput函数：

```
static double GetLeafSplitGainGivenOutput(double sum_gradients, double sum_hessians, double l1,
    const double sg_l1 = ThresholdL1(sum_gradients, l1);
    return -(2.0 * sg_l1 * output + (sum_hessians + l2) * output * output); // 叶子节点分数计算
  }
```

其中output代表叶子节点输出Wj 。考虑一般情况，假如叶子节点的样本一阶梯度和大于l1，那么，叶子节点分数的表达式为：

$$叶子节点分数 = -(2.0 * G_j * w_j + (H_j + l2) * w_j^2)$$

节点切分前的分数：

```
double gain_shift = GetLeafSplitGain(sum_gradient, sum_hessian,
                                      meta_->config->lambda_l1, meta_->config->lambda_l2,
                                      meta_->config->max_delta_step);
```

因此，计算节点切分增益：

节点切分增益= 节点分数 - （左叶子节点分数 + 右叶子节点分数）

比较切分增益与设置的最小增益大小，记为Gain：

Gain = 节点切分增益 — 最小切分增益

最后得到与XGBoost类似的形式，XGBoost为如下形式：

$$Gain = \frac{1}{2}[\underbrace{\frac{G_L^2}{H_L+\lambda}}_{\text{左子树分数}} + \underbrace{\frac{G_R^2}{H_R+\lambda}}_{\text{右子树分数}} - \underbrace{\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}}_{\text{分裂前分数}}] - \underbrace{\gamma}_{\text{新叶节点复杂度}}$$

LightGBM的最小切分增益对应于XGBoost的γ 。

2.直方图算法改进

直方图算法通过分段函数把连续值离散化成对应的bin的复杂度为O（#feature×#data），若降低feature或data的大小，那么直方图算法的复杂度也相应的降低，微软开源的LightGBM提供了两个算法分别降低feature和data的大小：

1. GOSS（减少样本角度）：保留梯度较大的样本数，减少梯度较小的样本个数 。样本训练误差小，表示该样本得到了很好的训练，对应的梯度亦越小 。一种直接的想法是抛弃这些梯度较小的样本，但是这种处理方法会改变样本集的分布，降低了训练模型的准确率（因为丢弃的都是损失误差较小的样本） 。因此，我们建议采用一种名为GOSS的方法，即GOSS保留具有大梯度的样本数，对梯度较小的样本进行随机采样，为了弥补数据集分布改变的影响，GOSS对小梯度的样本数增加了权重常数 。

GOSS伪代码如下：

Algorithm 2: Gradient-based One-Side Sampling

Input: I: training data, d: iterations
Input: a: sampling ratio of large gradient data
Input: b: sampling ratio of small gradient data
Input: $loss$: loss function, L: weak learner
models ← { }, fact ← $\frac{1-a}{b}$
topN ← a × len(I) , randN ← b × len(I)
for $i = 1$ ***to*** d **do**
  preds ← models.predict(I)
  g ← $loss$(I, preds), w ← {1,1,...}
  sorted ← GetSortedIndices(abs(g))
  topSet ← sorted[1:topN]
  randSet ← RandomPick(sorted[topN:len(I)], randN)
  usedSet ← topSet + randSet
  w[randSet] × = fact ▷ Assign weight $fact$ to the small gradient data.
  newModel ← L(I[usedSet], − g[usedSet], w[usedSet])
  models.append(newModel)

参考GOSS的解释，伪代码应该比较容易理解吧！

伪代码中损失函数的梯度g代表的含义是一阶梯度和二阶梯度的乘积，见Github的实现（LightGBM/src/boosting/goss.hpp）。

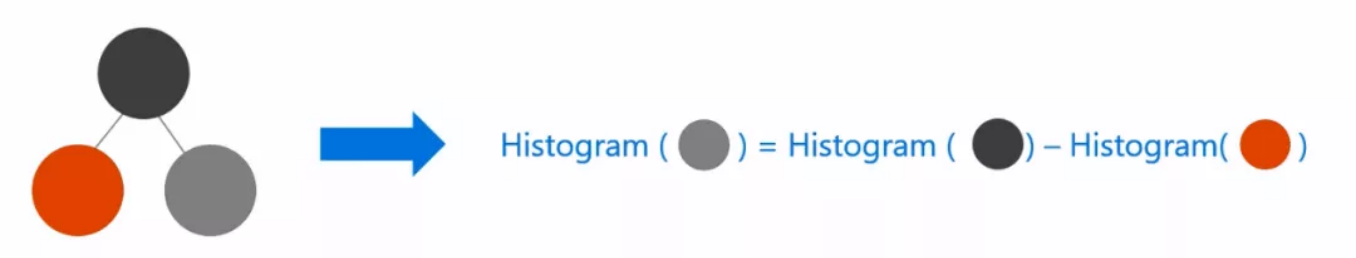
```
for (int cur_tree_id = 0; cur_tree_id < num_tree_per_iteration_; ++cur_tree_id) {
        size_t idx = static_cast<size_t>(cur_tree_id) * num_data_ + start + i;
        tmp_gradients[i] += std::fabs(gradients_[idx] * hessians_[idx]);  // 一阶梯度和二阶梯度乘
      }
```

2. EFB（从减少特征角度）：捆绑（bundling）互斥特征，用一个特征代替多个互斥特征，达到减少特征个数的目的 。EFB算法需要解决两个问题，（1）捆绑互斥特征，（2）合并互斥特征 。这里就不详细介绍了。

3. 直方图做差优化

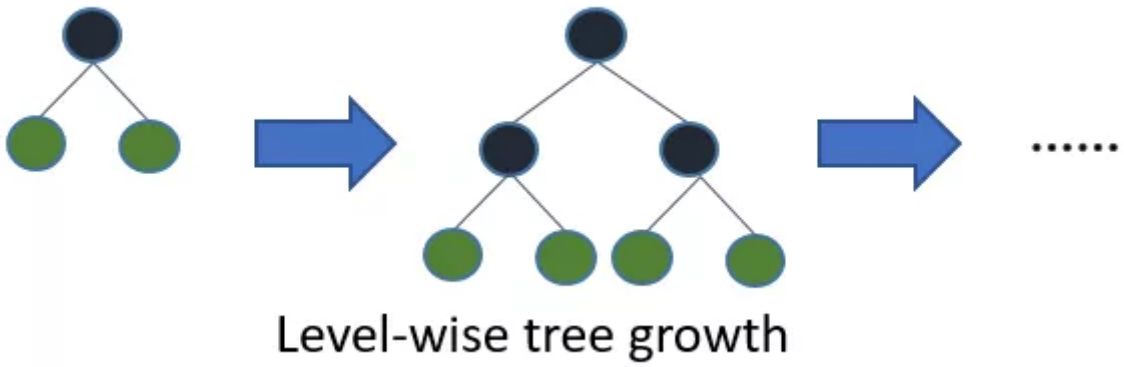
一个叶子节点的直方图可以由父亲节点的直方图与它兄弟的直方图做差得到 。利用这个方法，LightGBM可以用父亲节点的直方图减去数据量比较小的叶子节点直方图，得到数据量比较大的叶子节点直方图，因为该直方图是做差得到的，时间复杂度仅为O（#bins），比起不做差得到的兄弟节点的直方图，速度上可以提升一倍。

直方图做差示意图如下：

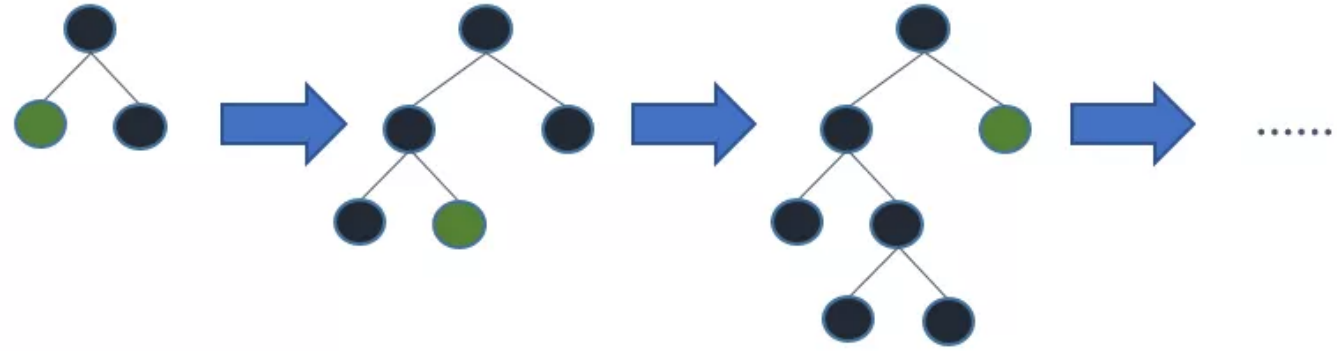


4. 树的生长策略

XGBoost是按层生长（level-wise）的方式展开节点，优点是不容易过拟合，缺点是它对每一层叶子节点不加区分的进行展开，实际上某些叶子节点的分裂增益较低，没必要进行搜索和分裂。如下图所示：



LightGBM是按最大增益的节点（叶子明智，Leaf-wise）进行展开，这样做的好处是找到分裂增益最大的叶子节点进行分裂，如此循环 。优点是效率高，在分裂次数相同的情况下，Leaf-wise可以得到更高的准确率 。缺点是可能会产生过拟合，通过设置树的最大生长深度避免 。如下图所示：



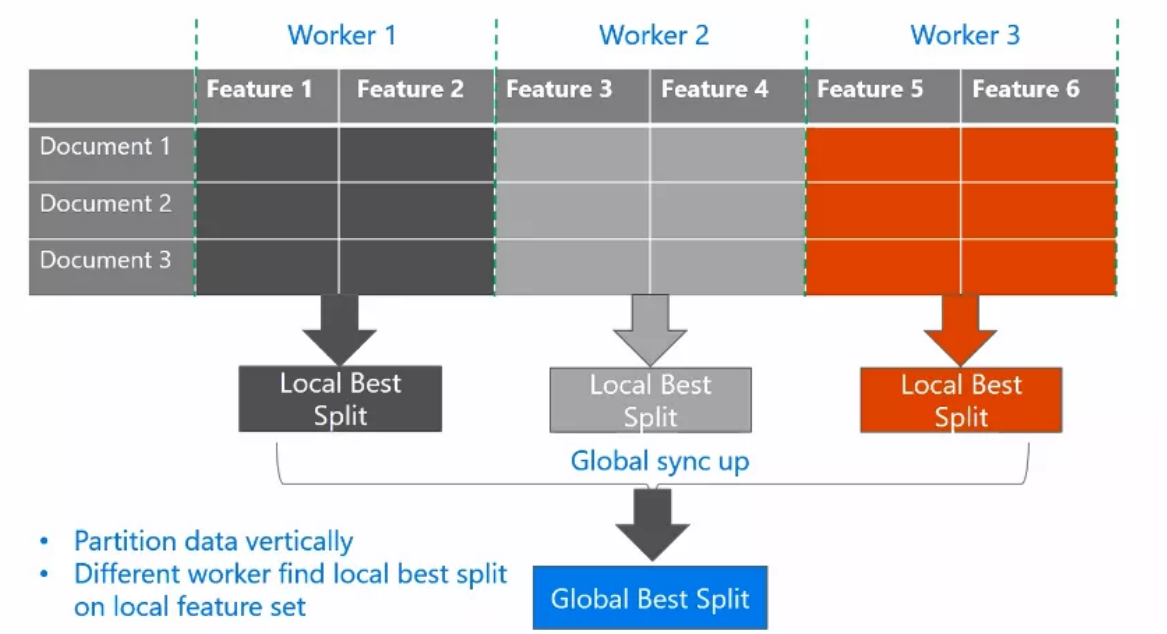
5. 支持类别特征

类别特征在实际项目中比较常见，XGBoost通过one-hot编码把类别特征转化为多维特征，降低了算法运行效率，LightGBM优化了类别特征的支持，不需要对类别特征进行one-hot编码 。

6. 支持高效并行

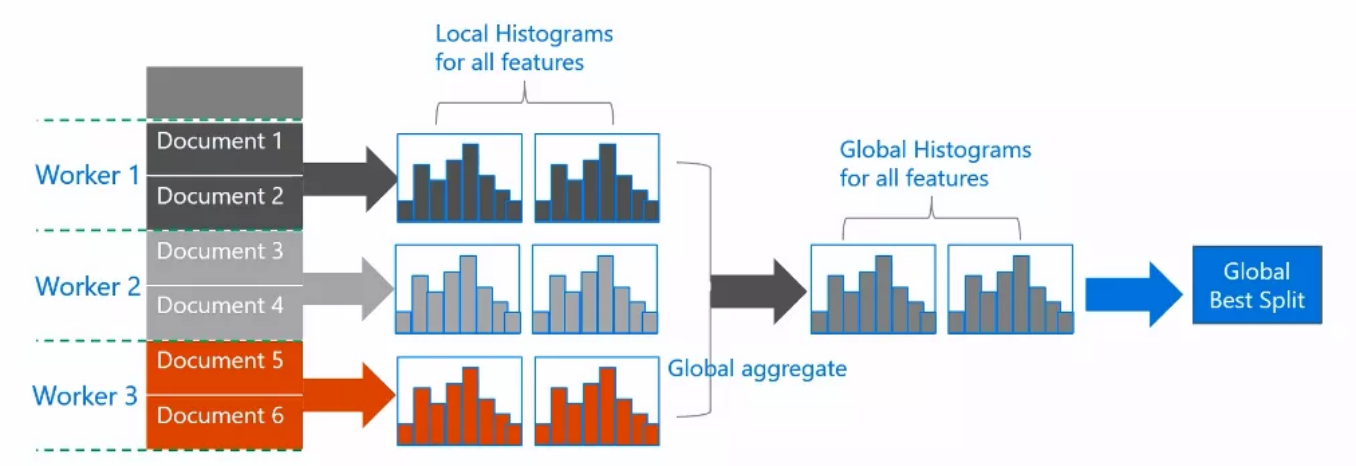
LightGBM支持特征并行和数据并行 。

特征并行：在不同机器选择局部不同特征对应的最优切分点，然后同步各机器结果，选择最优切分点 。如下图：



LightGBM对特征并行进行了优化，通过在本地保存全部数据避免对数据切分结果的通信 。

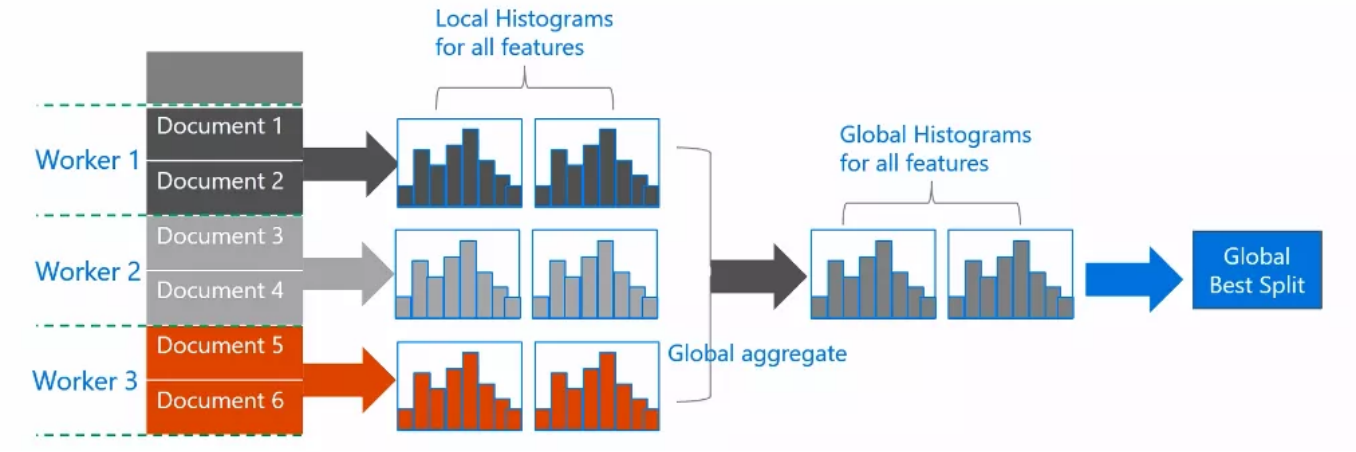
数据并行：不同机器在本地构造直方图，然后进行全局的合并，最后在合并的直方图上寻找最优分割点。如下图：



LightGBM有下面两种优化数据并行的方法：

（1）LightGBM通过"Reduce Scatter"将把直方图合并的任务分摊到不同的机器，降低通信和计算，并利用直方图做差，进一步减少了一半的通信量 。

（2）LightGBM通过“PV-Tree”的算法进行投票并行（Voting Parallel），使通信开销变成常数级别，在数据量很大的时候，使用投票并行可以得到非常好的加速效果。如下图：



7. 与XGBoost的对比

LightGBM相对于XGboost具有更快的训练效率和更低的内存使用，如下对比图：

	XGBoost	LightGBM
Tree growth algorithm	Level-wise good for engineering optimization but not efficient to learn model	Leaf-wise with max depth limitation get better trees with smaller computation cost, also can avoid overfitting
Split search algorithm	Pre-sorted algorithm	Histogram algorithm
memory cost	2*#feature*#data*4Bytes	#feature*#data*1Bytes (8x smaller)
Calculation of split gain	O(#data* #features)	O(#bin *#features)
Cache-line aware optimization	n/a	40% speed-up on Higgs data
Categorical feature support	n/a	8x speed-up on Expo data

LightGBM内存开销降低到原来的1/8倍 ， 在Expo数据集上速度快了8倍和在Higg数据集上加速了40%，

参考：

https://www.hrwhisper.me/machine-learning-lightgbm/

https://blog.csdn.net/huacha__/article/details/81057150

https://blog.csdn.net/anshuai_aw1/article/details/83040541

LightGBM: A Highly Efficient Gradient Boosting Decision Tree

推荐阅读

XGBoost算法原理小结

XGBoost参数调优小结

XGBoost切分点算法