王超（@德川） 陈帅华(@陈帅华 BigData)

笔者工作在汽车之家，在 kaggle 上 criteo 的点击率比赛中与陈天奇相识，autobots 在 700 多支 team 里排名第 7，模型用到了 fm 和 gbdt，其中 gbdt 当时使用的就是 xgboost，得见到其威力。

后续由于工作原因需要用到大规模机器学习的一些工具，而当时开源届并无太多成熟可靠方案。因此有幸参与到陈天奇的 yarn 版本的 xgboost 的开发过程，在这里备注一些心得供参考。

传统 GBDT 的理论推导

GBDT 模型全称 Gradient Boosted Decision Trees，在 1999 年由 Jerome Friedman 提出，将 GBDT 模型应用于 ctr 预估，最早见于 yahoo。

GBDT 是一个加性回归模型，通过 boosting 迭代的构造一组弱学习器，相对 LR 的优势如不需要做特征的归一化，自动进行特征选择，模型可解释性较好，可以适应多种损失函数如 SquareLoss，LogLoss 等等。但作为非线性模型，其相对线性模型的缺点也是显然的：boosting 是个串行的过程，不能并行化，计算复杂度较高，同时其不太适合高维稀疏特征，通常采用稠密的数值特征如点击率预估中的 COEC。

GBDT 模型在采用 LogLoss 时推导较逻辑回归复杂一些，我们这里给出具体原理和推导细节：

目标是寻找使得期望损失最小的决策函数 ，我们要求其具有一定的形式：即是一组弱学习器的加性组合。

$$F^* = argmin E_{x,y}[L(y, F(x))]$$
$$F(x; \rho_m, a_m) = \sum_{m=0}^{M} \rho_m h(x; a_m)$$

我们可以在函数空间上形式的使用梯度下降法求解，首先固定 x，对 F(x)求解其最优解。这里给出框架流程和 LogLoss 下的推导，一些变量的称谓沿用了原始 paper 里的叫法。

$$
\begin{aligned}
&F^*(x) = argminE_y[L(y, F(x))|x] \\
&F_0(x) = f_0(x) \\
&for \quad m = 1 \cdots M: \\
&\qquad g_m(x) = -\frac{\partial E_y[L(y, F(x))|x]}{\partial F(x)}|_{F(x)=F_{m-1}(x)} \qquad \text{descent direction} \\
&\qquad \rho_m = argminE_y[L(y, F_{m-1}(x) + \rho g_m(x)|x] \qquad \text{step size} \\
&\qquad f_m(x) = \rho_m g_m(x) \\
&\qquad F_m(x) = F_{m-1}(x) + \rho_m g_m(x) \\
&end \quad for \\
&F^*(x) \approx F_M(x) = f_0(x) + \sum_{m=1}^{M} \rho_m g_m(x)
\end{aligned}
$$

我们需要估计 g_m(x)，这里采用决策树的实现\beta_m h(x;a_m)去逼近函数 g_m(x)，使得俩者之间的距离尽可能的近。距离的衡量方式有很多选择，比如均方误差。这里给出 LogLoss 损失函数下的具体推导

$$
L(y, F) = log(1 + exp(-2yF)) \quad , \quad y \in \{-1, 1\}
$$

Step1.求解初始F_0。令其偏导为 0：

$$
\begin{aligned}
&F_0 = argmin \sum_{i=1}^{N} L(y_i, F) \\
&\frac{\partial \sum_{i=1}^{N} L(y_i, F)}{\partial F} = 0 \\
&\Rightarrow \sum_{i=1}^{N} \frac{exp\{-2y_iF\}(-2y_i)}{1 + exp\{-2y_iF\}} = 0 \\
&\Rightarrow \sum_{i:y_i=1} \frac{-2exp\{-2F\}}{1 + exp\{-2F\}} + \sum_{i:y_i=-1} \frac{2exp\{2F\}}{1 + exp\{2F\}} = 0 \\
&\Rightarrow F_0(x) = \frac{1}{2} log \frac{1 + \bar{y}}{1 - \bar{y}}
\end{aligned}
$$

Step2.估计 g_m(x)，并用决策树对其进行拟合:

$$
\begin{aligned}
g_m(x_i) \qquad &= -\frac{\partial L(y_i, F)}{\partial F}|_{F=F_{m-1}} \\
&= \frac{2y_i exp\{-2y_iF_{m-1}(x_i)\}}{1 + exp\{-2y_iF_{m-1}(x_i)\}} \\
&= 2y_i/(1 + exp\{2y_iF_{m-1}(x_i)\})
\end{aligned}
$$

Step3.用 a single Newton-Raphson step 去近似求解下降方向步长，通常的实现中 Step3 被省略，采用 shrinkage 的策略通过参数设置步长，避免过拟合

$$
\begin{aligned}
f(r) &= \sum_{x_i \in R_{jm}} log(1 + exp(-2y_i(F_{m-1}(x_i) + r))) \\
f'(r) &= \sum_{x_i \in R_{jm}} \frac{-2y_i}{1 + exp(2y_i(F_{m-1}(x_i) + r))} \\
f''(r) &= \sum_{x_i \in R_{jm}} \frac{2y_i exp(2y_i(F_{m-1}(x_i) + r)}{[1 + exp(2y_i(F_{m-1}(x_i) + r))]^2} \\
\gamma_{jm} &\approx \gamma_0 - f'(r_0)/f''(r_0) = \frac{\sum_{x_i \in R_{jm}} \tilde{y}_i}{\sum_{x_i \in R_{jm}} |\tilde{y}_i| \, (2 - |\tilde{y}_i|)}
\end{aligned}
$$

xgboost 版本的理论推导

不同于传统的 gbdt 方式，只利用了一阶的导数信息(上述 Step3 中 Newton-Raphson 会用到二阶信息，但一般实现中省略了 Step3)，xgboost 对 loss func 做了二阶的泰勒展开，并在目标函数之外加入了正则项整体求最优解，用以权衡目标函数的下降和模型的复杂程度，避免过拟合。具体推导详见陈天奇的 ppt，这里给出简要的摘注，一些变量的称谓沿用陈天奇 ppt 里的叫法，和前述 friedman 的版本里不一致，请注意。

将目标函数做泰勒展开，并引入正则项：

$$
\begin{aligned}
&Obj^{(t)} = \sum_{i=1}^{n} l(y_i, y_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant \\
&\approx \sum_{i=1}^{n} \left[l(y_i, y_i^{(t-1)}) + \partial_{y_i^{(t-1)}} l(y_i, y_i^{(t-1)}) f_t(x_i) + \frac{1}{2} \partial^2_{y_i^{(t-1)}} l(y_i, y_i^{(t-1)}) f_t^2(x_i) \right] \\
&+ \Omega(f_t) + constant
\end{aligned}
$$

除去常数项，求得每个样本的一阶导 g_i 和二阶导 h_i，将目标函数按叶子节点规约分组，略去一些中间步骤:

$$
\begin{aligned}
Obj^{(t)} &\approx \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \\
&= \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma T
\end{aligned}
$$

在树结构是 fix 的时候，上式中叶子节点权重 w_j 有闭式解，解和对应的目标函数值如下:

$$
\begin{aligned}
w_j^* &= -\frac{G_j}{H_j+\lambda} \\
Obj &= \frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda}+\gamma T
\end{aligned}
$$

在目标函数是 LogLoss 损失函数下，这里给出一阶导 g_i 和二阶导 h_i 的推导：

$$
\begin{aligned}
&g_i = \partial_{y_i^{(t-1)}} l(y_i, y_i^{(t-1)}) \\
&= -y_i\left(1-\frac{1}{1+e^{-y_i^{(t-1)}}}\right)+(1-y_i)\frac{1}{1+e^{-y_i^{(t-1)}}} \\
&= Pred - Label
\end{aligned}
$$

$$
\begin{aligned}
&h_i = \partial^2_{y_i^{(t-1)}} l(y_i, y_i^{(t-1)}) = \frac{e^{-y_i^{(t-1)}}}{\left(1+e^{-y_i^{(t-1)}}\right)^2} \\
&= Pred*(1-Pred)
\end{aligned}
$$

其他细节见后续代码部分的解读。

分布式 xgboost 的设计理念

除去理论上和传统 gbdt 的差别外，从使用者的角度，xgboost 的设计理念在使用时主要有如下几点感受：

1．速度快

让一个程序在必要的时候占领一台机器，并且在所有迭代的时候一直跑到底，来防止重新分配资源的开销。

机器内部采用单机多线程方式来并行加速匀速，机器之间通信基于 rabit 实现的 all reduce 的同步接口。

2．可移植，少写代码

大部分的分布式机器学习算法的结构都是分布数据，在每个子集上面算出一些局部的统计量，然后整合出全局的统计量，并且在分配给各个计算节点去进行下一轮的迭代。

根据算法本身的需求，抽象出合理的接口如 Allreduce，并通过通用的库如 rabit

让平台往接口需求上面去走，最终使得各种比较有效的分布式机器学习 abstraction 的实现在各个平台下面跑。

这里按自己理解给出一个整体框架的理念的分层抽象逻辑。

<table>
<tr><td colspan="2">Gbdt</td><td colspan="2">Lr</td><td>…</td></tr>
<tr><td colspan="5">Interface abstraction(AllReduce,异步 SGD,…)</td></tr>
<tr><td>Yarn</td><td>MPI</td><td colspan="2">Openmp</td><td>…</td></tr>
</table>

3．可容错

Rabit 版本的 Allreduce 有一个很好的性质，支持容错，而传统的 mpi 是不支持的。具体实现方式：Allreduce 每一个节点最后拿到的是一样的结果，这意味着可以让一些节点记住结果。当有节点挂掉重启的时候，可以直接向还活着的节点索要结果就可以了。

分布式 xgboost 发展过程

Xgboost 的分布式算法开始就基本成熟了，分布式版本的变更主要是底层的 allreduce 接口支持不同平台[mpi,yarn,…]的过程。

在这里主要简述一下支持 yarn 版本的分布式 xgboost 的发展历程，主要开发工作是陈天奇完成，我们小组做了一些配合性质的工作。

第一版支持 MPI

Rabit 开始兼容 mpi 的 allreduce 但是运行 mpi 的分布式程序需要读写 hdfs 并分割数据。当时没有实现 C++读写 hdfs 功能，跑 hdfs 数据比较麻烦。

第二版支持 Hadoop Streaming

Hadoop 版 通过 hadoop streaming 把 xgboost 作为 map 运行。在每个 map 中进行 allreduce 的通信完 xgboost 每轮的迭代，reduce 只是负责输出模型。通过在一个 map 中完成 xgboost 所有的迭代，避免了传统基于 hadoop 的机器学习程序每轮迭代间的资源的重新分配，使得基于 hadoop 的 xgboost 的迭代速度能媲美基于 mpi 的 xgboost，同时避免了 mpi 程序读写 hdfs 的麻烦。

第三版原生支持 Yarn

Yarn 为了能自由的掌控各个节点的资源分配并且能使文件均衡的分割到各个节点，直接把 rabit 作为 yarn 的一个 APP 运行。在这一版里，实现了 c++读写 hdfs 的功能，解决了 streaming 版本中 xgboost 中对文件操作自由不高的问题。

分布式通信框架 rabit 简述

rabit 是容错的 allreduce 实现，在启动 yarn 集群基于 rabit 的任务时，首先在节点间建立树形的连接关系（下图）来提高节点间的通信效率。建立连接关系后，各节点可通过 allreduce 接口通信内存单元。

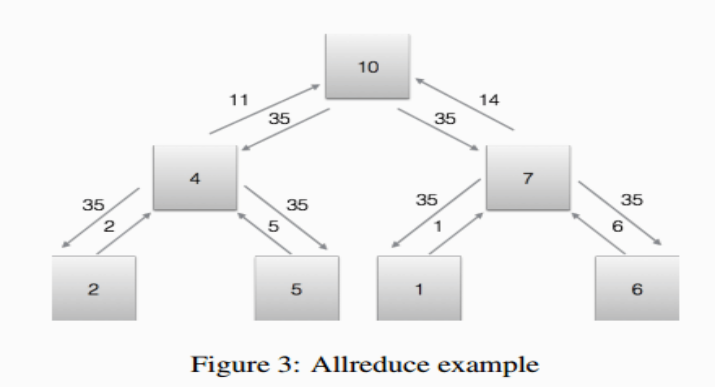

Figure 3: Allreduce example

上图是 rabit 中 allreduce 求和的例子。每个节点依次向父节点发送节点值，父节点接受子节点的值并和父节点自己的值求和后，再往上一层节点发送。最后根节点得到全部子节点的和并和自己的节点值相加后就得到了所有节点的和，然后将根节点求得的值沿着树的路径传播给各个节点。allreduce 的求和操作完成后，各个节点得到了所有节点值的和。

rabit allreduce 支持对连续内存单元简单的 sum max min 操作，也支持自定义的 reduce 操作来对不同节点中的类的数据成员合并。xgboost 的分布式算法主要用到了自定义的 reduce 操作来合并计算特征候选分割点的统计量，以及合并由候选分割点得到的区间内样本的统计量。

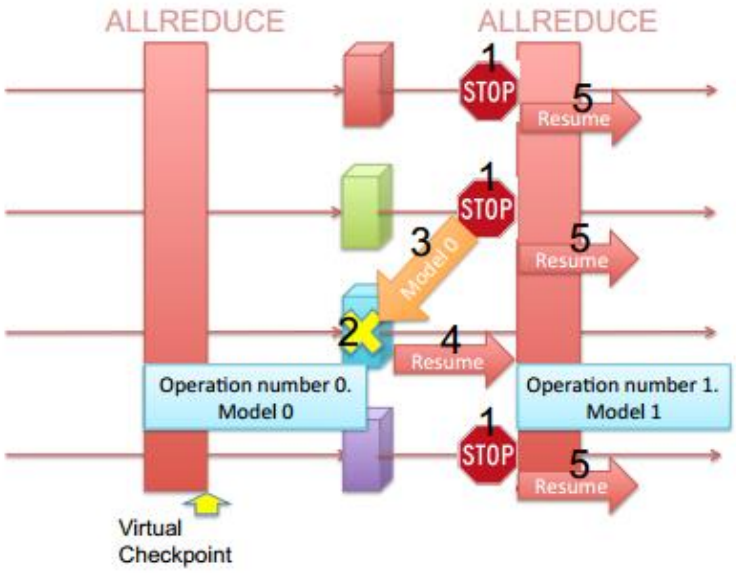

Figure 4: Fault Tolerance

rabit 设计的 allreduce 还具有容错功能：从上图中可以看到，rabit 在一轮 allreduce 通信结束后会在各节点内存中将模型结果缓存为检查点，并增加检查点版本号。同步结果后，各节点会继续各自的计算直到下一次 allreduce 通信同步，在这个过程中如果其中有节点发生故障计算失败，该故障节点会从树形连接的集群中找到最近的节点，拿到上一轮的模型文件，然后重新开始计算。其他无故障节点等待故障节点直到故障节点计算完成后，各节点间才再一次进行 allreduce 通信同步。
在分布式 xgboost 中，使用检查点在迭代建树过程中保存模型，使得 xgboost 在模型更新过程中具有容错能力。

xgboost 代码简析

xgboost 源码目录结构

我们将源码结构中和 yarn 版本相关的部分代码抽离出来，先简要描述一下每个文件的功用。

```
├──── data.h  基础数据结构如 bst_gpair,sparse_batch,row_batch,IFMatrix
├──── gbm
│  ├──── gbm.cpp  IGradBooster 接口工厂类
│  ├──── gbm.h  IGradBooster 接口
│  └──── gbtree-inl.hpp  gbdt 具体实现
├──── io
│  ├──── io.cpp
│  ├──── io.h  主要对 siple_dmatrix-inl 的 LoadBinary 和 SaveBinary 方法进行封装，加载时，可以从二进制、文本文件以及 cache 文件（.buffer）中加载数据，保存文件时，都会保存为二进制文件
│  ├──── simple_dmatrix-inl.hpp  对各种来源的输入数据进行转换，最终存入 MetaInfo 结构中，以及一些对 MetaInfo 结构结构的操作函数
│  └──── simple_fmatrix-inl.hpp  将数据以稀疏矩阵形式进行操作以及存取
├──── learner
│  ├──── dmatrix.h MetaInfo(样本 label，权重等数据集信息) DMatrix(数据集+特征矩阵的信息)
│  ├──── evaluation.h 评估 auc,loss 等接口
│  ├──── evaluation-inl.hpp 评估的具体实现
│  ├──── helper_utils.h  LogSum,SoftMax 等数学变换
│  ├──── learner-inl.hpp  gbdt 的训练
│  ├──── objective.h  目标函数计算梯度等的接口和相关工厂类
│  └──── objective-inl.hpp  具体优化目标实现，回归,分类,排序
├──── sync
│  └──── sync.h 封装 raibt 同步相关的头文件
├──── tree
│  ├──── model.h  TreeModel 和 RegTree 的数据结构定义
│  ├──── param.h  在 tree 的命名空间里定义了 TrainParam，GradStats，CVGradStats，SplitEntry 数据结构
│  ├──── updater_basemaker-inl.hpp 构建树算法的基类
│  ├──── updater.cpp  update 接口的工厂类
│  ├──── updater.h  RegTree 的 update 接口
│  ├──── updater_histmaker-inl.hpp 构建树
│  ├──── updater_prune-inl.hpp 剪枝树后同步
```

```
│   └──── updater_sync-inl.hpp 同步树实现
├──── utils
│   ├──── config.h 配置管理
│   ├──── fmap.h 特征名和特征类型
│   ├──── group_data.h 构建 CSR/CSC 矩阵的构造器
│   ├──── io.h 调用 rabit 的 IO 接口
│   ├──── iterator.h 迭代器虚基类
│   ├──── matrix_csr.h CSR 矩阵操作
│   ├──── omp.h openMP 接口
│   ├──── quantile.h 分位数统计结构如 WQSummary,WXQSummary,GKSummary,以及
QuantileSketchTemplate 算法接口
│   ├──── random.h random 库
│   └──── utils.h print 等基础函数
└──── xgboost_main.cpp
```

目标函数接口设计

在 objective.h 中同样的采用了工厂类的设计模式，内置定义了各种常用的损失函数，方便根据业务需求来扩展优化目标。一般的开源实现大多仅仅支持 0-1 分类或者回归，但 xgboost 的实现便于后续扩展，目前已经支持了包括多分类还有排序和泊松回归的各种目标函数，后续读者还可以插拔自定义的其他损失函数。

在配置文件里目前可以自定义优化目标，目前版本包括以下优化目标：

```
inline IObjFunction* CreateObjFunction(const char *name) {
    using namespace std;
    if (!strcmp("reg:linear", name)) return new RegLossObj(LossType::
kLinearSquare);
    if (!strcmp("reg:logistic", name)) return new RegLossObj(LossType
::kLogisticNeglik);
    if (!strcmp("binary:logistic", name)) return new RegLossObj(LossT
ype::kLogisticClassify);
    if (!strcmp("binary:logitraw", name)) return new RegLossObj(LossT
ype::kLogisticRaw);
    if (!strcmp("count:poisson", name)) return new PoissonRegression(
);
    if (!strcmp("multi:softmax", name)) return new SoftmaxMultiClassO
bj(0);
    if (!strcmp("multi:softprob", name)) return new SoftmaxMultiClass
Obj(1);
    if (!strcmp("rank:pairwise", name )) return new PairwiseRankObj();
    if (!strcmp("rank:ndcg", name)) return new LambdaRankObjNDCG();
    if (!strcmp("rank:map", name)) return new LambdaRankObjMAP();
    return NULL;
}
```

树更新策略接口设计

在 updater.h 中采用了工厂模式，已经内置定义了各种 updater 继承 update 接口去修改树。

```
IUpdater* CreateUpdater(const char *name) {
  if (!strcmp(name, "prune")) return new TreePruner();
  if (!strcmp(name, "refresh")) return new TreeRefresher<GradStats>();
  if (!strcmp(name, "grow_colmaker")) return new ColMaker<GradStats>(
);
  if (!strcmp(name, "sync")) return new TreeSyncher();
  if (!strcmp(name, "grow_histmaker")) return new CQHistMaker<GradSta
ts>();
  if (!strcmp(name, "grow_skmaker")) return new SketchMaker();
  if (!strcmp(name, "distcol")) return new DistColMaker<GradStats>();
  return NULL;
}
```

这些 update 操作又可以互相组合，支持串联多个 updater 去更新树，给后续带来比较大的可扩展性。

```
std::string tval = tparam.updater_seq;
char *pstr;
pstr = std::strtok(&tval[0], ",");
while (pstr != NULL) {
  updaters.push_back(tree::CreateUpdater(pstr));
  ……
  pstr = std::strtok(NULL, ",");
}
```

比如在 row based 的分布式 xgboost 实现中，tparam.updater_seq 被初始化为 "grow_histmaker,prune"，实现先更新树，后剪枝的功能。又比如可以有多种 grow 策略，然后用同一类去 prune。

xgboost 启动过程

Yarn、MPI 和 Sungrid Engine 集群上启动 dmlc job 的脚本结构如下图，tracker 支持启动的 dmlc job 包括基于 rabit allreduce 和 parameter server 的程序。分布式 xgboost 启动的 dmlc job 是基于 rabit allreduce 的程序。

```
├─── tracker
|       ├─── dmlc_local.py 本地测试脚本
|       ├─── dmlc_mpi.py  在 mpi 集群上启动 dmlc job
```

```
|       ├──── dmlc_yarn.py 在 yarn 集群上启动 dmlc job
|       ├──── dmlc_sge.py 在 Sungrid Engine 集群上启动 dmlc job
|       ├─── tracker.py 按树形结构建立集群通信连接关系并管理 allreduce 迭代过程，管理 parameter server 通信过程。
```

```
|── yarn
    ├──── src
    |       └──── org
    |            └──── apache
    |                 └──── hadoop
    |                      └──── yarn 下面是 yarn 集群资源申请和参数设置
    |                           └──── dmlc 的代码
    |                                ├──── ApplicationMaster.java
    |                                ├──── Client.java
    |                                └──── TaskRecord.java
    ├──── run_hdfs_prog.py 设置 c++读写 hdfs 文件的 libhdfs 库的环境变量
```

下面是在 yarn 集群上启动基于 rabit all reduce 通信的分布式 xgboost 的任务脚本，dmlc_yarn.py 后面输入参数包括集群设置参数和具体运行的 xgboost dmlc job 命令。

```
../../dmlc-core/tracker/dmlc_yarn.py -n $1 --vcores $2 ../../xgboost.dmlc mushroom.hadoop.conf nthread=$2\
    data=hdfs://$3/data/agaricus.txt.train\
    eval[test]=hdfs://$3/data/agaricus.txt.test\
    model_out=hdfs://$3/mushroom.final.model
```

dmlc_yarn.py 启动的 dmlc job 包括 rabit 和 paremeter server。 由于分布式 xgboost 是基于 rabit allreduce 的，这里介绍 dmlc_yarn.py 中启动 rabit 任务的部分。

dmlc_yarn.py 负责集群参数解析并生成最终的 yarn 任务的运行命令，传到 tracker 中的 submit 方法中，在 submit 方法中建立节点的树形连接，然后向 yarn 集群申请资源，获得资源后分布式 xgboost 在每轮建树过程中 allreduce 通信来计算特征分割点，直到达到设定的棵数，所有迭代的 allreduce 通信完成，rabit 任务结束。

```
def submit(nworker, nserver, fun_submit, hostIP = 'auto', pscmd = None):
    //初始化 RabitTracker，各节点建立通信连接
    rabit = RabitTracker(hostIP = hostIP, nslave = nworker)
    //各节点启动 rabit 任务，各节点间建立树形连接
    rabit.start(nworker)
    //各节点运行提交的程序，节点间按 allreduce 通信
    fun_submit(nworker, nserver, envs)
```

分布式加载数据

src/io/io.cpp 中的 LoadDataMatrix 函数，训练集从 hdfs 地址加载数据，解析成 DataMatrix 类型的数据

```
DataMatrix* LoadDataMatrix(const char *fname,
                        bool silent,
                        bool savebuffer,
                        bool loadsplit,
                        const char *cache_file) {
  DMatrixSimple *dmat = new DMatrixSimple();
  //从 hdfs 加载数据，解析数据格式
  dmat->LoadText(fname, silent, loadsplit);
  return dmat;
}
```

src/io/simple_dmatrix-inl.hpp 具体实现分布式加载

```
void LoadText(const char *uri, bool silent = false, bool loadsplit = false)
{
   //按输入的 hdfs 地址分割数据并按节点的 rank 编号给每个节点分配分割后的数据
   LibSVMParser   parser(dmlc::InputSplit::Create(uri,   rank,   npart,
"text"), 16);
   //libsvm 格式解析数据，存到 DMatrixSimple 对应的 CSR 的数据结构中
   while (parser.Next()) {
     const LibSVMPage &batch = parser.Value();
     .......
   }
}
```

分布式训练

GBDT 训练主流程在 src/xgboost_main.cpp 中，按设定的树棵树每轮迭代新建回归树。为了使得模型训练能够容错，每轮迭代中，在模型建树和评估当前模型结束后，增加模型版本号并设立模型的检查点。

```
// GBDT 训练迭代过程
 inline void TaskTrain(void) {
   for (int i = version / 2; i < num_round; ++i) {
       // 每轮训练
      if (version % 2 == 0) {
        //模型新建一棵树
        learner.UpdateOneIter(i, *data);
```

```
        //allreduce 各节点同步模型，支持容错
        rabit::CheckPoint(&learner);
       // 模型的版本号+1
        version += 1;
      }
     //每轮验证集评估
      string res = learner.EvalOneIter(i,devalall, eval_data_names);
      rabit::CheckPoint(&learner);
      version += 1;
  }
}
```

具体每棵树的建树更新流程在 src/learner/learner-inl.hpp 中，logloss 样本一阶导和二阶导见理论部分推导

```
//每轮建树迭代
 inline void UpdateOneIter(int iter, const DMatrix &train) {
     // 计算样本模型预测值
    this->PredictRaw(train, &preds_);
    //计算样本一阶和二阶导数
    obj_->GetGradient(preds_, train.info, iter, &gpair_);
    //boost 建树
    gbm_->DoBoost(train.fmat(),this->FindBufferOffset(train),
    train.info.info, &gpair_);
  }
```

Boost 流程在 src/gbm/gbtree-inl.hpp 中

```
virtual void DoBoost(IFMatrix *p_fmat,
                     int64_t buffer_offset,
                     const BoosterInfo &info,
                     std::vector<bst_gpair> *in_gpair) {
     const std::vector<bst_gpair> &gpair = *in_gpair;
     std::vector<std::vector<tree::RegTree*> > new_trees;
     //以二分类为列，num_output_group=1 是二分类
     if (mparam.num_output_group == 1) {
        new_trees.push_back(BoostNewTrees(gpair, p_fmat, buffer_offset,
info, 0));
       //将新生成的树追加到模型中
       this->CommitModel(new_trees[0], 0);
     }
}

inline std::vector<tree::RegTree*>
BoostNewTrees(const std::vector<bst_gpair> &gpair,
```

```
                IFMatrix *p_fmat,
                int64_t buffer_offset,
                const BoosterInfo &info,
                int bst_group) {
    std::vector<tree::RegTree *> new_trees;
    //根据配置实例化对应的 updater
    this->InitUpdater();
    //新建树并按设置的参数初始化树，GBDT num_parallel_tree=1，随机森林等参数取值大于 1
    for (int i = 0; i < tparam.num_parallel_tree; ++i) {
      new_trees.push_back(new tree::RegTree());
      for (size_t j = 0; j < cfg.size(); ++j) {
        new_trees.back()->param.SetParam(cfg[j].first.c_str(), cfg[j].second.c_str());
      }
      new_trees.back()->InitModel();
    }
    //串联多个 updater，row based 的分布式 xgboostgrow 树时用到的 updater 是 updater_histmaker-inl.hpp 中的 CQHistMaker<GradStats>()
    for (size_t i = 0; i < updaters.size(); ++i) {
      updaters[i]->Update(gpair, p_fmat, info, new_trees);
    }
    return new_trees;
}
```

按 row based 进行分布式的建树流程在 src/tree/updater_histmaker-inl.hpp 中，每台机器各自 propose 各自的候选分割点，每一部分算出自己的统计量，用 allreduce 合并起来后再根据全局统计量计算最终的分割点，最终层次遍历的构建树

```
virtual void Update(const std::vector<bst_gpair> &gpair,
                    IFMatrix *p_fmat,
                    const BoosterInfo &info,
                    RegTree *p_tree) {

    //初始化每条样本对应的树节点的 position，初始化层级遍历对应的树节点队列 qexpand
    this->InitData(gpair, *p_fmat, info.root_index, *p_tree);

    //多机并行计算每个特征的最大最小值并同步，初始化特征集合 fwork_set
    this->InitWorkSet(p_fmat, *p_tree, &fwork_set);
```

```
    //层次遍历构建单棵树
    for (int depth = 0; depth < param.max_depth; ++depth) {
      //不同于传统的抽样+排序的生成候选分割点的策略，这里的实现基于可并行化的近似直方图算法，大幅加速生成分割点的效率，具体算法待陈天奇 paper 公开
      this->ResetPosAndPropose(gpair, p_fmat, info, fwork_set, *p_tree);
      //多线程按特征粒度并行对每个候选分割桶的一阶二阶导求和，并最终通过 Allreduce 多机求和同步统计量,用于后续计算全局分割点
      this->CreateHist(gpair, p_fmat, info, fwork_set, *p_tree);
      //按待更新树节点粒度并行，根据 loss 下降程度计算每个树节点的最佳 split，分割该节点或设为叶子节点，更新树结构
      this->FindSplit(depth, gpair, p_fmat, info, fwork_set, p_tree);
      //按样本粒度并行，更新各样本所属的树节点
      this->ResetPositionAfterSplit(p_fmat, *p_tree);
      //更新层次遍历的树节点队列
      this->UpdateQueueExpand(*p_tree);
      if (qexpand.size() == 0) break;
    }
    //更新叶子节点权重
    for (size_t i = 0; i < qexpand.size(); ++i) {
      const int nid = qexpand[i];
      (*p_tree)[nid].set_leaf(p_tree->stat(nid).base_weight * param.learning_rate);
    }
}
```

xgboost 实战

汽车之家的广告业务中用到了点击率预估的技术，在 kaggle 的比赛中，我们应用了一些比 LR 效果更好的非线性模型取得了不错的成绩(如 gbdt,fm)，希望将这些技术应用在公司的实际场景中。由于广告的展示日志数据规模很大，需要用到分布式的模型训练工具。

我们并没有 MPI 集群，如果要采用 gbdt，虽然模型效果最好的当时有 xgboost 的单机多线程版本，但并行版本可行的方案当时只有 spark 的 gbdt 刚有雏形。在采用了 spark 上的 gbdt 一段时间后，我们发现其中效率和效果上的一些问题，而且 spark 的社区在 gbdt 这块的迭代并不是很快，于是萌生了和陈天奇一起把 xgboost 在 yarn 上并行化，更好的推广到工业界上的想法。

参数调参

这里简介一下采用 yarn 版本分布式 xgboost 调参的经验，供后来者参考。具体的调参还需要根据自有业务，在理解 xgboost 实现的基础上，自行调优。

由于点击率预估的正样本比较稀疏，假设平均 CTR py 很小（比如 0.001），有两个重要的量 g 梯度，h 二阶导，g= py - y，h = py(1-py)，这个时候 h 一般也比较小。参考理论推导一节，会导致叶子节点权重比较大，进而导致每一步的估计 step 过大。

这时有几个参数可以调节一下：

min_child_weight

这个参数默认是 1，是每个叶子里面 h 的和至少是多少，对正负样本不均衡时的 0-1 分类而言，假设 h 在 0.01 附近，min_child_weight 为 1 意味着叶子节点中最少需要包含 100 个样本。这个参数非常影响结果，控制叶子节点中二阶导的和的最小值，该参数值越小，越容易 overfitting。

eta [learning_rate 同义]

shrinkage 参数，用于更新叶子节点权重时，乘以该系数，避免步长过大。参数值越大，越可能无法收敛。把学习率 eta 设置的小一些，小学习率可以使得后面的学习更加仔细。附上涉及的代码：

```
(*p_tree)[nid].set_leaf(p_tree->stat(nid).base_weight * param.learning_rate);
```

scale_pos_weight

如果优化的是仅仅展示排序，就是 AUC 的话，可以采用平衡正负样本权重的办法调大正样本权重。设置 scale_pos_weight 就可以把正样本权重乘这个系数。如果还需要优化回归的性能，还需要在此基础上做下 recalibration。

max_delta_step

如果设立了该值，对叶子节点的权重值做了约束在[max_delta_step, max_delta_step]。以防在某些 loss 下权重值过大，默认是 0（其实代表 inf）。可以试试把这个参数设置到 1-10 之间的一个值。这样会防止做太大的更新步子，使得更新更加平缓。

由于这里对权重做了修正，不是闭式解，对应的目标函数值也回退到了原始的算法。附上和 max_delta_step 相关的部分核心代码，涉及公式请参照 xgboost 的理论推导一节：

```
//计算目标函数值
inline double CalcGain(double sum_grad, double sum_hess) const {
    if (sum_hess < min_child_weight) return 0.0;
    if (max_delta_step == 0.0f) {
      if (reg_alpha == 0.0f) {
```

```
            return Sqr(sum_grad) / (sum_hess + reg_lambda);
        } else {
            return Sqr(ThresholdL1(sum_grad, reg_alpha)) / (sum_hess + reg_lambda);
        }
    } else {
        double w = CalcWeight(sum_grad, sum_hess);
        double ret = sum_grad * w + 0.5 * (sum_hess + reg_lambda) * Sqr(w);
        if (reg_alpha == 0.0f) {
            return - 2.0 * ret;
        } else {
            return - 2.0 * (ret + reg_alpha * std::abs(w));
        }
    }
}
//计算叶子节点权重
inline double CalcWeight(double sum_grad, double sum_hess) const {
    if (sum_hess < min_child_weight) return 0.0;
    double dw;
    if (reg_alpha == 0.0f) {
        dw = -sum_grad / (sum_hess + reg_lambda);
    } else {
        dw = -ThresholdL1(sum_grad, reg_alpha) / (sum_hess + reg_lambda);
    }
    if (max_delta_step != 0.0f) {
        if (dw > max_delta_step) dw = max_delta_step;
        if (dw < -max_delta_step) dw = -max_delta_step;
    }
    return dw;
}
```

这里还涉及到了另外俩个参数，这里一并介绍一下：

lambda [reg_lambda 同义]

控制模型复杂程度的权重值的 L2 正则项参数，参数值越大，模型越不容易 overfitting。

alpha [reg_alpha 同义]

控制模型复杂程度的权重值的 L1 正则项参数，参数值越大，模型越不容易 overfitting。

gamma [min_split_loss 同义]

后剪枝时，用于控制是否后剪枝的参数，附上相关代码

```
inline bool need_prune(double loss_chg, int depth) const
{
    return loss_chg < this->min_split_loss;
}
```

max_depth

gbdt 每颗树的最大深度，树高越深，越容易过拟合。

num_round

gbdt 的棵数，棵数越多，训练误差越小，但是棵数过多容易过拟合。需要同时观察训练 loss 和测试 loss，确定最佳的棵数。

验证集 loss 打印

xgboost 支持 eval 的数据从 hdfs 读入，所以在训练时可以同步看一下 eval-logloss，对比一下不同参数情况下结果的情况。

最后需要补充一下的是，模型整体比较鲁棒，这也是为什么 xgboost 在众多 kaggle 比赛里很出彩的原因之一。大多数情况下，只需要调节 max_depth,num_round,min_child_weight,eta 就能取得不错的效果。

常见问题

1.running beyond virtual memory limits 错误

跑大的数据，需要设置每台节点分配的内存 通过-mem 设置

2.服务器 gcc 版本不支持 c++11，无法编译 wormhole，而且没有权限修改 /usr/lib，/usr/lib64 和/etc/ld.so.conf

安装新版本 gcc 后 把 wormle 中的 gcc 更换成新版本地址。接下来可以直接把新版本的 libstdc++.a 和 xgboost 静态连接。或者将新版本 libstdc++.so.6.0.18 拷贝到脚本目录并重命名为 libstdc++.so.6。然后在 yarn/run_hdfs_prog.py 中给 lpath 添加当前目录./，之后通过-f 命令缓存脚本目录下的动态库。

3.xgboost 在迭代过程中，跑到某一棵树的时候一直卡在这里，kill job 之后查看 log,发现报 too many large chunk 错误

这一般是由于正样本比较稀疏，导致每次估计的叶子权重比较大，可以调低 min_child_weight 和 eta,或者将 max_delta_step 设 1-10 之间的值。

速度测试

不方便在这里透露实际训练集的情况，在这里仅以简单的采样后的数据做的 bias model 为例来测试一下 xgboost 的性能(bias model 指的是只加入 bias feature 如广告位等，以不加入排序相关用户特征)

训练集　62G　验证集 22G
Xgboost 参数 深度设 4　跑 50 棵树　max_delta_step=5
集群参数　-mem　15000　节点 25　线程 20
总时间：1338s　其中数据分割和加载：145s　每棵树迭代时间　大约 24s