Alink漫谈(十三)： 在线学习算法FTRL 之 具体实现

原创 罗西的思考 罗西的思考 7月22日

Alink漫谈(十三)： 在线学习算法FTRL 之 具体实现

[Toc]

0x00 摘要

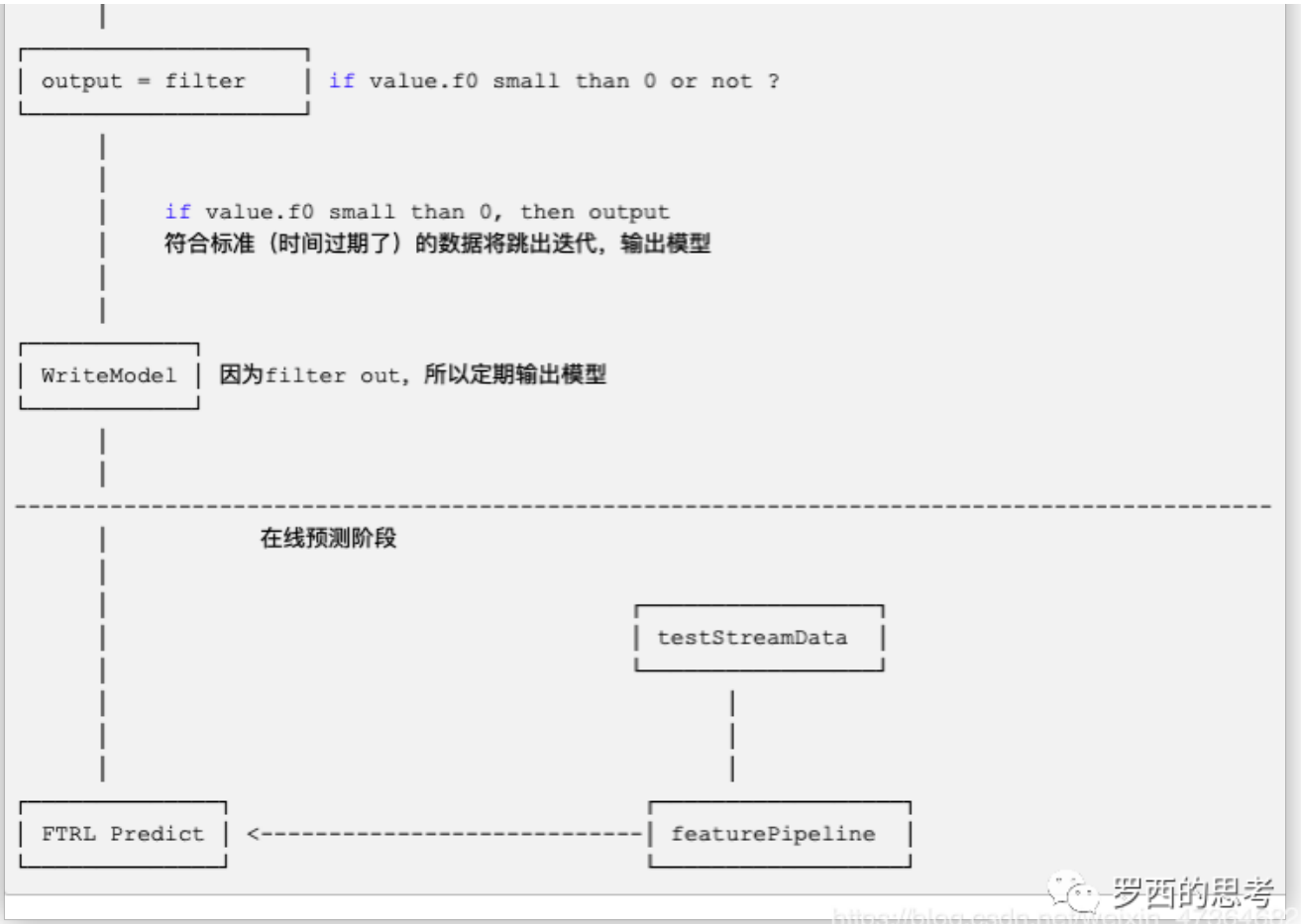
Alink 是阿里巴巴基于实时计算引擎 Flink 研发的新一代机器学习算法平台，是业界首个同时支持批式算法、流式算法的机器学习平台。本文和上文一起介绍了在线学习算法 FTRL 在Alink中是如何实现的，希望对大家有所帮助。

0x01 回顾

书接上回 Alink漫谈(十二)： 在线学习算法FTRL 之 整体设计 。到目前为止，已经处理完毕输入，接下来就是在线训练。训练优化的主要目标是找到一个方向，参数朝这个方向移动之后使得损失函数的值能够减小，这个方向往往由一阶偏导或者二阶偏导各种组合求得。

为了让大家更好理解，我们再次贴出整体流程图：


初始模型训练阶段
trainBatchData
trainStreamData
featurePipeline
线性回归模型
特征向量
SplitVector
解析输入，得到DataStream<Tuple5<SampleId, taskId, numSubVec, SubVec, label>> input
<Tuple5,Tuple7> iteration
迭代构建，两个输入train data Tuple5<>, feedback data Tuple7<>
CalcTask从逻辑上分成两个模块：flatMap1, flatMap2
CalcTask.flatMap1
输入Tuple5<>
CalcTask.flatMap2
输入Tuple7
分布计算FTRL算法中的predict部分
分布处理反馈数据/更新参数/累积参数到期后发出
以上两个flatmap都输出到下面ReduceTask
ReduceTask.flatMap
1. 如果时间过期&全部收集完成，归并/输出模型(value.f0 < 0)
2. 未过期，归并每个CalcTask计算的predict，形成一个 lable y
result = filter
if t3.f0 > 0 && t3.f2 > 0 or not ?
if t3.f0 > 0 && t3.f2 > 0 then
CalcTask.flatMap2
输出Tuple7
"时间未过期&向量有意义" 将送回反馈，继续迭代
如果未形成反馈数据流，则继续过滤



整体流程

0x02 在线训练

在线训练主要逻辑是：

- 1）加载初始化模型到 dataBridge；dataBridge = DirectReader.collect(model);
- 2）获取相关参数。比如vectorSize默认是30000，是否 hasInterceptItem；
- 3）获取切分信息。splitInfo = getSplitInfo(featureSize, hasInterceptItem, parallelism); 下面马上会用到。
- 4）切分高维向量。初始化数据做了特征哈希，会产生高维向量，这里需要进行切割。initData.flatMap(new SplitVector(splitInfo, hasInterceptItem, vectorSize,vectorTrainIdx, featureIdx, labelIdx))；
- 5）构建一个 IterativeStream.ConnectedIterativeStreams iteration，这样会构建（或者说连接）两个数据流：反馈流和训练流；
- 6）用iteration来构建迭代体 iterativeBody，其包括两部分：CalcTask，ReduceTask；
- 6.1）CalcTask分成两个部分。flatMap1 是分布计算FTRL迭代需要的predict，flatMap2 是FTRL的更新参数部分；

- 6.2）ReduceTask分为两个功能：“归并这些predict计算结果“ / ”如果满足条件则归并模型 & 向下游算子输出模型“；
- 7）result = iterativeBody.filter；基本是以时间间隔为标准来判断（也可以认为是时间驱动），"时间未过期&向量有意义" 的数据将被发送回反馈数据流，继续迭代，**回到步骤 6)，进入flatMap2**；
- 8）output = iterativeBody.filter；符合标准（时间过期了）的数据将跳出迭代，然后算法会调用WriteModel将LineModelData转换为多条Row，转发给下游operator（也就是在线预测阶段）；**即定时把模型更新给在线预测阶段**。

2.1 预置模型

前面说到，FTRL先要训练出一个**逻辑回归模型**作为FTRL算法的初始模型，这是为了系统冷启动的需要。

2.1.1 训练模型

具体逻辑回归模型设定/训练是：

```
// train initial batch model
LogisticRegressionTrainBatchOp lr = new LogisticRegressionTrainBatchOp()
            .setVectorCol(vecColName)
            .setLabelCol(labelColName)
            .setWithIntercept(true)
            .setMaxIter(10);
BatchOperator<?> initModel = featurePipelineModel.transform(trainBatchData).link(lr);
```

训练好之后，模型信息是DataSet类型，位于变量 BatchOperator initModel之中，这是一个批处理算子。

2.1.2 加载模型

FtrlTrainStreamOp将initModel作为初始化参数。

```
FtrlTrainStreamOp model = new FtrlTrainStreamOp(initModel)
```

在FtrlTrainStreamOp构造函数中会加载这个模型；

```
dataBridge = DirectReader.collect(initModel);
```

具体加载时通过MemoryDataBridge直接获取初始化模型DataSet中的数据。

```
public MemoryDataBridge generate(BatchOperator batchOperator, Params globalParams) {
   return new MemoryDataBridge(batchOperator.collect());
}
```

2.2 分割高维向量

从前文可知，Alink的FTRL算法设置的特征向量维度是30000。所以算法第一步就是切分高维度向量，以便分布式计算。

```
String vecColName = "vec";
int numHashFeatures = 30000;
```

首先要获取切分信息，代码如下，就是将特征数目featureSize 除以 并行度parallelism，然后得到了每个task对应系数的初始位置。

```
private static int[] getSplitInfo(int featureSize, boolean hasInterceptItem, int paral
    int coefSize = (hasInterceptItem) ? featureSize + 1 : featureSize;
    int subSize = coefSize / parallelism;
    int[] poses = new int[parallelism + 1];
    int offset = coefSize % parallelism;
    for (int i = 0; i < offset; ++i) {
        poses[i + 1] = poses[i] + subSize + 1;
    }
    for (int i = offset; i < parallelism; ++i) {
        poses[i + 1] = poses[i] + subSize;
    }
    return poses;
}
//程序运行时变量如下
featureSize = 30000
hasInterceptItem = true
parallelism = 4
coefSize = 30001
subSize = 7500
poses = {int[5]@11660}
 0 = 0
 1 = 7501
 2 = 15001
 3 = 22501
 4 = 30001
offset = 1
```

然后根据切分信息对高维向量进行切割。

```
// Tuple5<SampleId, taskId, numSubVec, SubVec, label>
DataStream<Tuple5<Long, Integer, Integer, Vector, Object>> input
          = initData.flatMap(new SplitVector(splitInfo, hasInterceptItem, vectorSize,
                            vectorTrainIdx, featureIdx, labelIdx))
          .partitionCustom(new CustomBlockPartitioner(), 1);
```

具体切分在SplitVector.flatMap函数完成，结果就是把一个高维度向量分割给各个CalcTask。

代码摘要如下：

```
public void flatMap(Row row, Collector<Tuple5<Long, Integer, Integer, Vector, Object>>
                long sampleId = counter;
        counter += parallelism;
        Vector vec;
        if (vectorTrainIdx == -1) {
            .....
        } else {
            // 输入row的第vectorTrainIdx个field就是那个30000大小的系数向量
            vec = VectorUtil.getVector(row.getField(vectorTrainIdx));
        }

        if (vec instanceof SparseVector) {
            Map<Integer, Vector> tmpVec = new HashMap<>();
            for (int i = 0; i < indices.length; ++i) {
              .....
              // 此处迭代完成后，tmpVec中就是task number个元素，每一个元素是分割好的系数向量
            }
            for (Integer key : tmpVec.keySet()) {
                //此处遍历，给后面所有CalcTask发送五元组数据。
                collector.collect(Tuple5.of(sampleId, key, subNum, tmpVec.get(key), row
            }
        } else {
         ......
        }
    }
}
```

这个Tuple5.of(sampleId, key, subNum, tmpVec.get(key), row.getField(labelIdx))就是后面CalcTask的输入。

2.3 迭代训练

此处理论上有以下几个重点：

- 预测方法：在每一轮t中，针对特征样本xt，以及迭代后（第一次则是给定初值）的模型参数wt，我们可以预测该样本的标记值：pt=σ(wt,xt)，其中σ(a)=1/(1+exp(−a))是一个sigmoid函数。

- 损失函数：对一个特征样本xt，其对应的标记为yt ∈ 0,1，则通过 logistic loss 来作为损失函数。

- 迭代公式：我们的目的是使得损失函数尽可能的小，即可以采用极大似然估计来求解参数。首先求梯度，然后使用FTRL进行迭代。

伪代码思路大致如下

```
double p = learner.predict(x); //预测
learner.updateModel(x, p, y);  //更新模型
double loss = LogLossEvalutor.calLogLoss(p, y); //计算损失
evalutor.addLogLoss(loss); //更新损失
totalLoss += loss;
trainedNum += 1;
```

具体实施上Alink有自己的特点和调整。

2.3.1 Flink Stream迭代功能

机器学习都需要迭代训练，Alink这里利用了Flink Stream的迭代功能。

IterativeStream的实例是通过DataStream的iterate方法创建的˙。iterate方法存在两个重载形式：

- 一种是无参的，表示不限定最大等待时间；
- 一种提供一个长整型maxWaitTimeMillis参数，允许用户指定等待反馈边的下一个输入元素的最大时间间隔。

Alink选择了第二种。

在创建ConnectedIterativeStreams时候，用迭代流的初始输入作为第一个输入流，用反馈流作为第二个输入。

每一种数据流（DataStream）都会有与之对应的流转换（StreamTransformation）。IterativeStream对应的转换是FeedbackTransformation。

迭代流（IterativeStream）对应的转换是反馈转换（FeedbackTransformation），它表示拓扑中的一个反馈点（也即迭代头）。一个反馈点包含一个输入边以及若干个反馈边，且Flink要求每个反馈边的并行度必须跟输入边的并行度一致，这一点在往该转换中加入反馈边时会进行校验。

当IterativeStream对象被构造时，FeedbackTransformation的实例会被创建并传递给DataStream的构造方法。

迭代的关闭是通过调用IterativeStream的实例方法closeWith来实现的。这个函数指定了某个流将成为迭代程序的结束，并且这个流将作为输入的第二部分（second input）被反馈回迭代。

2.3.2 迭代构建

对于Alink来说，迭代构建代码是：

```
// train data format = <sampleId, subSampleTaskId, subNum, SparseVector(subSample), lab
// feedback format = Tuple7<sampleId, subSampleTaskId, subNum, SparseVector(subSample),
IterativeStream.ConnectedIterativeStreams<
    Tuple5<Long, Integer, Integer, Vector, Object>,
    Tuple7<Long, Integer, Integer, Vector, Object, Double, Long>>
    iteration = input.iterate(Long.MAX_VALUE)
    .withFeedbackType(TypeInformation
    .of(new TypeHint<Tuple7<Long, Integer, Integer, Vector, Object, Double, Long>>() {}

// 即iteration是一个 IterativeStream.ConnectedIterativeStreams<...>
```

2.3.2.1 迭代的输入

从代码和注释可以看出，迭代的两种输入是：

- train data format = <sampleId, subSampleTaskId, subNum, SparseVector(subSample), label>; 这种其实是训练数据;
- Tuple7<sampleId, subSampleTaskId, subNum, SparseVector(subSample), label, wx, timeStamps>; 这种其实是反馈数据，就是“迭代的反馈流”作为这个第二输入 (second input);

2.3.2.2 迭代的反馈

反馈流的设置是通过调用IterativeStream的实例方法closeWith来实现的。Alink这里是

```
DataStream<Tuple7<Long, Integer, Integer, Vector, Object, Double, Long>>
        result = iterativeBody.filter(
            return (t3.f0 > 0 && t3.f2 > 0); // 这里是省略版本代码
            );

iteration.closeWith(result);
```

前面已经提到过，result filter 的判断是 `return (t3.f0 > 0 && t3.f2 > 0)` ，如果满足条件，则说明时间未过期&向量有意义，所以此时应该反馈回去，继续训练。

反馈流的格式是：

- Tuple7<sampleId, subSampleTaskId, subNum, SparseVector(subSample), label, wx, timeStamps>;

2.3.3 迭代体 CalcTask / ReduceTask

迭代体由两部分构成：CalcTask / ReduceTask。

CalcTask每一个实例都拥有初始化模型dataBridge。

```
DataStream iterativeBody = iteration.flatMap(
    new CalcTask(dataBridge, splitInfo, getParams()))
```

2.3.3.1 迭代初始化

迭代是由 CalcTask.open 函数开始，主要做如下几件事

- 设定各种参数，比如
  - 工作task个数，numWorkers = getRuntimeContext().getNumberOfParallelSubtasks();
  - 本task的id，workerId = getRuntimeContext().getIndexOfThisSubtask();
- 读取初始化模型
  - List<Row> modelRows = DirectReader.directRead(dataBridge);
  - 把 Row 类型数据转换为线性模型 LinearModelData model = new LinearModelDataConverter().load(modelRows);
- 读取本task对应的系数 coef[i - startIdx]，这里就是把整个模型切分到numWorkers这么多的Task中，并行更新。
- 指定本task的开始时间 startTime = System.currentTimeMillis();

2.3.3.2 处理输入数据

CalcTask.flatMap1主要实现的是FTRL算法中的predict部分（注意，不是FTRL预测）。

解释：pt=σ(Xt·w)是LR的预测函数，求出pt的唯一目的是为了求出目标函数（在LR中采用交叉熵损失函数作为目标函数）对参数w的一阶导数g，gi=(pt−yt)xi。此步骤同样适用于FTRL优化其他目标函数，唯一的不同就是求次梯度g（次梯度是左导和右导之间的集合，函数可导--左导等于右导时，次梯度就等于一阶梯度）的方法不同。

函数的输入是 "训练输入数据"，即`SplitVector.flatMap的输出 ----> CalcCalcTask的输入`。输入数据是一个五元组，其格式为 train data format = <sampleId, subSampleTaskId, subNum, SparseVector(subSample), label>;

有三点需要注意：

- 是如果是第一次进入，则需要savedFristModel；
- 这里是有输入就处理，然后立即输出（和flatMap2不同，flatMap2有输入就处理，但不是立即输出，而是当时间到期了再输出）；
- predict的实现： `((SparseVector)vec).getValues()[i] * coef[indices[i] - startIdx];`

大家会说，不对！predict函数应该是 `sigmoid = 1.0 / (1.0 + np.exp(-w.dot(x)))` 。是的，这里还没有做 sigmoid 操作。当ReduceTask做了聚合之后，会把聚合好的 p 反馈回迭代体，然后在 CalcTask.flatMap2 中才会做 sigmoid 操作。

```
public void flatMap1(Tuple5<Long, Integer, Integer, Vector, Object> value,
                     Collector<Tuple7<Long, Integer, Integer, Vector, Object, Double, L
    if (!savedFristModel) { //第一次进入需要存模型
        out.collect(Tuple7.of(-1L, 0, getRuntimeContext().getIndexOfThisSubtask(),
            new DenseVector(coef), labelValues, -1.0, modelId++));
        savedFristModel = true;
    }
    Long timeStamps = System.currentTimeMillis();
    double wx = 0.0;
    Long sampleId = value.f0;
    Vector vec = value.f3;
    if (vec instanceof SparseVector) {
        int[] indices = ((SparseVector)vec).getIndices();
        // 这里就是具体的Predict
        for (int i = 0; i < indices.length; ++i) {
            wx += ((SparseVector)vec).getValues()[i] * coef[indices[i] - startIdx];
        }
    } else {
       ......
    }
    //处理了就输出
    out.collect(Tuple7.of(sampleId, value.f1, value.f2, value.f3, value.f4, wx, timeSta
}
```

2.3.3.3 归并数据

ReduceTask.flatMap 负责归并数据。

```
public static class ReduceTask extends
    RichFlatMapFunction<Tuple7<Long, Integer, Integer, Vector, Object, Double, Long>,
        Tuple7<Long, Integer, Integer, Vector, Object, Double, Long>> {
    private int parallelism;
    private int[] poses;
    private Map<Long, List<Object>> buffer;
    private Map<Long, List<Tuple2<Integer, DenseVector>>> models = new HashMap<>();
}
```

flatMap函数大致完成如下功能，即两种归并：

- 为了输出模型使用。判断是否时间过期 if (value.f0 < 0)，如果过期，则归并模型：
  - 生成一个 List<Tuple2<Integer, DenseVector>> model = models.get(value.f6); 以value.f6，即时间戳为key，插入到HashMap中。
  - 如果全部收集完成，则向下游算子输出模型，并且从HashMap中删除暂存的模型。
- 为了归并predict使用。归并每个CalcTask计算的predict，形成一个 lable y；
  - 用 label y 更新 Tuple7 的 f5， 即 Tuple7<sampleId, subSampleTaskId, subNum, SparseVector(subSample), label, wx, timeStamps> 中的 label，也就是预测的 y。
  - 给每个下游算子（就是每个CalcTask了，不过是作为flatMap2的输入）发送这个新 Tuple7；

当具体用作输出模型使用时，其变量如下：

```
models = {HashMap@13258}  size = 1
 {Long@13456} 1 -> {ArrayList@13678}  size = 1
  key = {Long@13456} 1
  value = {ArrayList@13678}  size = 1
   0 = {Tuple2@13698} "(1,0.0 -8.244533295515879E-5 0.0 -1.103997743166529E-4 0.0 -3.33
```

2.3.3.4 判断是否反馈

这个 filter result 是用来判断是否反馈的。这里t3.f0 是sampleId, t3.f2是subNum。

```
DataStream<Tuple7<Long, Integer, Integer, Vector, Object, Double, Long>>
    result = iterativeBody.filter(
    new FilterFunction<Tuple7<Long, Integer, Integer, Vector, Object, Double, Long>>()
        @Override
        public boolean filter(Tuple7<Long, Integer, Integer, Vector, Object, Double, Lo
            throws Exception {
            // if t3.f0 > 0 && t3.f2 > 0 then feedback
            return (t3.f0 > 0 && t3.f2 > 0);
        }
    });
```

对于 **t3.f0**，有两处代码会设置为负值。

- 会在savedFirstModel 这里设置一次"-1"；即

```
    if (!savedFristModel) {
          out.collect(Tuple7.of(-1L, 0, getRuntimeContext().getIndexOfThisSubtask(),
                      new DenseVector(coef), labelValues, -1.0, modelId++));
      savedFristModel = true;
    }
```

- 也会在时间过期时候设置为 "-1"。

```
if (System.currentTimeMillis() - startTime > modelSaveTimeInterval) {
  startTime = System.currentTimeMillis();
  out.collect(Tuple7.of(-1L, 0, getRuntimeContext().getIndexOfThisSubtask(),
      new DenseVector(coef), labelValues, -1.0, modelId++));
}
```

对于 **t3.f2**，如果 subNum 大于零，说明在高维向量切分时候，是得到了有意义的数值。

因此 `return (t3.f0 > 0 && t3.f2 > 0)` 说明时间未过期&向量有意义，所以此时应该反馈回去，继续训练。

2.3.3.5 判断是否输出模型

这里是filter output。

`value.f0 < 0` 说明时间到期了，应该输出模型。

```
DataStream<Row> output = iterativeBody.filter(
    new FilterFunction<Tuple7<Long, Integer, Integer, Vector, Object, Double, Long>>()
        @Override
        public boolean filter(Tuple7<Long, Integer, Integer, Vector, Object, Double, Lo
        {
            /* if value.f0 small than 0, then output */
            return value.f0 < 0;
        }
    }).flatMap(new WriteModel(labelType, getVectorCol(), featureCols, hasInterceptItem)
```

2.3.3.6 处理反馈数据/更新参数

CalcTask.flatMap2实际完成的是FTRL算法的其余部分，即更新参数部分。主要逻辑如下：

- 计算时间间隔 timeInterval = System.currentTimeMillis() - value.f6;
- 正式计算predict， p = 1 / (1 + Math.exp(-p)); 即sigmoid 操作;
- 计算梯度 g = (p - label) * values[i] / Math.sqrt(timeInterval); 这里除以了时间间隔;
- 更新参数;
- 输入。注意，这里是有输入就处理，但 **不是立即输出**，而是累积参数，当时间到期了再输出，也就是做到了定期输出模型;

在 **Logistic Regression** 中， sigmoid函数是σ(a) = 1 / (1 + exp(-a))， 预估 pt = σ(xt . wt), 则 LogLoss 函数是

$$l_t(w_t) = -y_t log(p_t) - (1 - y_t) lo$$

直接计算可以得到

$$\nabla l(w) = (\sigma(w.x_t) - y_t)x_t = ($$

具体 LR + FTRL 算法实现如下：

```
@Override
public void flatMap2(Tuple7<Long, Integer, Integer, Vector, Object, Double, Long> value
                     Collector<Tuple7<Long, Integer, Integer, Vector, Object, Double, L
    throws Exception {
    double p = value.f5;
    // 计算时间间隔
    long timeInterval = System.currentTimeMillis() - value.f6;
    Vector vec = value.f3;

    /* eta */
    // 正式计算predict，之前只是计算了一半，这里计算后半部，即
    p = 1 / (1 + Math.exp(-p));
    .....

    if (vec instanceof SparseVector) {
        // 这里是更新参数
        int[] indices = ((SparseVector)vec).getIndices();
        double[] values = ((SparseVector)vec).getValues();

        for (int i = 0; i < indices.length; ++i) {
            // update zParam nParam
            int id = indices[i] - startIdx;
            // values[i]是xi
            // 下面的计算基本和Google伪代码一致
            double g = (p - label) * values[i] / Math.sqrt(timeInterval);
            double sigma = (Math.sqrt(nParam[id] + g * g) - Math.sqrt(nParam[id])) / al
            zParam[id] += g - sigma * coef[id];
            nParam[id] += g * g;

            // update model coefficient
            if (Math.abs(zParam[id]) <= l1) {
                coef[id] = 0.0;
            } else {
                coef[id] = ((zParam[id] < 0 ? -1 : 1) * l1 - zParam[id])
                    / ((beta + Math.sqrt(nParam[id]) / alpha + l2));
            }
        }
    } else {
      ......
    }

    // 当时间到期了再输出，即做到了定期输出模型
    if (System.currentTimeMillis() - startTime > modelSaveTimeInterval) {
        startTime = System.currentTimeMillis();
        out.collect(Tuple7.of(-1L, 0, getRuntimeContext().getIndexOfThisSubtask(),
```

```
            new DenseVector(coef), labelValues, -1.0, modelId++));
    }
}
```

2.4 输出模型

WriteModel 类实现了输出模型功能，大致逻辑如下：

- 生成一个LinearModelData，用训练好的Tuple7来填充这个 LinearModelData。其中两个重要点：
  - modelData.coefVector = (DenseVector)value.f3;
  - modelData.labelValues = (Object[])value.f4;
- 把模型数据转换成List rows。LinearModelDataConverter().save(modelData, listCollector);
- 序列化，发送给下游算子。因为模型可能会很大，所以这里打散之后分布发送给下游算子。

```
public void flatMap(Tuple7<Long, Integer, Integer, Vector, Object, Double, Long> value,

//输入value变量打印如下：
value = {Tuple7@13296}
 f0 = {Long@13306} -1
 f1 = {Integer@13307} 0
 f2 = {Integer@13308} 2
 f3 = {DenseVector@13309} "-0.7383426732137565 0.0 0.0 0.0 1.5885293675862715E-4 -4.834
  data = {double[30001]@13314}
 f4 = {Object[2]@13310}
 f5 = {Double@13311} -1.0
 f6 = {Long@13312} 0

        //生成模型
        LinearModelData modelData = new LinearModelData();
        ......
        modelData.coefVector = (DenseVector)value.f3;
        modelData.labelValues = (Object[])value.f4;

        //把模型数据转换成List<Row> rows
        RowCollector listCollector = new RowCollector();
        new LinearModelDataConverter().save(modelData, listCollector);
        List<Row> rows = listCollector.getRows();

        for (Row r : rows) {
            int rowSize = r.getArity();
            for (int j = 0; j < rowSize; ++j) {
                        .....
              //序列化
            }
            out.collect(row);
```

```
            }

            iter++;
        }
    }
```

0x03 在线预测

预测功能是在 FtrlPredictStreamOp 完成的。

```
// ftrl predict
FtrlPredictStreamOp predictResult = new FtrlPredictStreamOp(initModel)
        .setVectorCol(vecColName)
        .setPredictionCol("pred")
        .setReservedCols(new String[]{labelColName})
        .setPredictionDetailCol("details")
        .linkFrom(model, featurePipelineModel.transform(splitter.getSideOutput(0)));
```

从上面代码我们可以看到

- FtrlPredict 功能同样需要初始模型 initModel，我们也是把逻辑回归模型赋予它。这样也是为了冷启动，即当FTRL训练模块还没有产生模型之前，FTRL预测模块也是可以对其输入数据做预测的。
- model 是 FtrlTrainStreamOp 的输出，即 FTRL 的训练输出。所以 WriteModel 就直接把输出传给了 FtrlPredict功能。
- splitter.getSideOutput(0) 这里是前面提到的测试输入，就是测试数据集。

linkFrom函数完成了业务逻辑，大致功能如下：

- 使用 `inputs[0].getDataStream().flatMap ------> partition ----> map ----> flatMap(new CollectModel())` 得到了模型 LinearModelData modelstr；
- 使用 DataStream.connect 把输入的测试数据集 和 模型 LinearModelData modelstr关联起来，这样每个 task 都拥有了在线模型 modelstr，就可以通过 `flatMap(new PredictProcess(…)` 进行分布式预测；
- 使用 setOutputTable 和 LinearModelMapper 把预测结果输出；

即 FTRL的预测功能有三个输入：

- **初始模型 initModel** -----> 最后被 PredictProcess.open 加载，作为冷启动的预测模型；
- **测试数据流** -----> 被 PredictProcess.flatMap1处理，进行预测；

- **FTRL训练阶段产生的模型数据流** ----> 被 PredictProcess.flatMap2 处理，进行在线模型更新；

3.1 初始化

构造函数中完成了初始化，即获取事先训练好的逻辑回归模型。

```
public FtrlPredictStreamOp(BatchOperator model) {
    super(new Params());
    if (model != null) {
        dataBridge = DirectReader.collect(model);
    } else {
        throw new IllegalArgumentException("Ftrl algo: initial model is null. Please se
    }
}
```

3.2 获取在线训练模型

CollectModel完成了 获取在线训练模型 功能。

其逻辑主要是：模型被分成若干块，其中 (long)inRow.getField(1) 这里记录了具体有多少块。所以 flatMap 函数会把这些块累积起来，最后组装成模型，统一发送给下游算子。

具体是通过一个 HashMap<> buffers 来完成临时拼装/最后组装的。

```
public static class CollectModel implements FlatMapFunction<Row, LinearModelData> {

    private Map<Long, List<Row>> buffers = new HashMap<>(0);

    @Override
    public void flatMap(Row inRow, Collector<LinearModelData> out) throws Exception {

// 输入参数如下
inRow = {Row@13389} "0,19,0,{"hasInterceptItem":"true","vectorCol":"\"vec\"","modelName
 fields = {Object[5]@13405}
  0 = {Long@13406} 0
  1 = {Long@13403} 19
  2 = {Long@13406} 0
  3 = "{"hasInterceptItem":"true","vectorCol":"\"vec\"","modelName":"\"Logistic Regress

        long id = (long)inRow.getField(0);
        Long nTab = (long)inRow.getField(1);

        Row row = new Row(inRow.getArity() - 2);
```

```
        for (int i = 0; i < row.getArity(); ++i) {
            row.setField(i, inRow.getField(i + 2));
        }

        if (buffers.containsKey(id) && buffers.get(id).size() == nTab.intValue() - 1) {
            buffers.get(id).add(row);
            // 如果累积完成，则组装成模型
            LinearModelData ret = new LinearModelDataConverter().load(buffers.get(id));
            buffers.get(id).clear();
            // 发送给下游算子。
            out.collect(ret);
        } else {
            if (buffers.containsKey(id)) {
                //如果有key。则往list添加。
                buffers.get(id).add(row);
            } else {
                // 如果没有key，则添加list
                List<Row> buffer = new ArrayList<>(0);
                buffer.add(row);
                buffers.put(id, buffer);
            }
        }
    }
}

//变量类似这种
this = {FtrlPredictStreamOp$CollectModel@13388}
 buffers = {HashMap@13393}  size = 1
  {Long@13406} 0 -> {ArrayList@13431}  size = 2
   key = {Long@13406} 0
    value = 0
   value = {ArrayList@13431}  size = 2
    0 = {Row@13409} "0,{"hasInterceptItem":"true","vectorCol":"\"vec\"","modelName":"\'
    1 = {Row@13471} "1048576,{"featureColNames":null,"featureColTypes":null,"coefVector
```

3.3 在线预测

PredictProcess 完成了在线预测功能，LinearModelMapper 是具体预测实现。

```
public static class PredictProcess extends RichCoFlatMapFunction<Row, LinearModelData,
    private LinearModelMapper predictor = null;
    private String modelSchemaJson;
    private String dataSchemaJson;
    private Params params;
    private int iter = 0;
    private DataBridge dataBridge;
}
```

3.3.1 加载预设置模型

其构造函数获得了 FtrlPredictStreamOp 类的 dataBridge，即事先训练好的逻辑回归模型。每一个Task都拥有完整的模型。

open函数会加载逻辑回归模型。

```
public void open(Configuration parameters) throws Exception {
    this.predictor = new LinearModelMapper(TableUtil.fromSchemaJson(modelSchemaJson),
        TableUtil.fromSchemaJson(dataSchemaJson), this.params);
    if (dataBridge != null) {
        // read init model
        List<Row> modelRows = DirectReader.directRead(dataBridge);
        LinearModelData model = new LinearModelDataConverter().load(modelRows);
        this.predictor.loadModel(model);
    }
}
```

3.3.2 在线预测

FtrlPredictStreamOp.flatMap1 函数完成了在线预测。

```
public void flatMap1(Row row, Collector<Row> collector) throws Exception {
    collector.collect(this.predictor.map(row));
}
```

调用栈如下：

```
predictWithProb:157, LinearModelMapper (com.alibaba.alink.operator.common.linear)
predictResultDetail:114, LinearModelMapper (com.alibaba.alink.operator.common.linear)
map:90, RichModelMapper (com.alibaba.alink.common.mapper)
flatMap1:174, FtrlPredictStreamOp$PredictProcess (com.alibaba.alink.operator.stream.onl
flatMap1:143, FtrlPredictStreamOp$PredictProcess (com.alibaba.alink.operator.stream.onl
processElement1:53, CoStreamFlatMap (org.apache.flink.streaming.api.operators.co)
processRecord1:135, StreamTwoInputProcessor (org.apache.flink.streaming.runtime.io)
```

具体是通过 LinearModelMapper 完成。

```
public abstract class RichModelMapper extends ModelMapper {
    public Row map(Row row) throws Exception {
        if (isPredDetail) {
            // 我们的示例代码在这里
            Tuple2<Object, String> t2 = predictResultDetail(row);
            return this.outputColsHelper.getResultRow(row, Row.of(t2.f0, t2.f1));
        } else {
            return this.outputColsHelper.getResultRow(row, Row.of(predictResult(row)));
```

```
        }
    }
}
```

预测代码如下，可以看出来使用了sigmoid。

```
/**
 * Predict the label information with the probability of each label.
 */
public Tuple2 <Object, Double[]> predictWithProb(Vector vector) {
   double dotValue = MatVecOp.dot(vector, model.coefVector);
   switch (model.linearModelType) {
      case LR:
      case SVM:
         double prob = sigmoid(dotValue);
         return new Tuple2 <>(dotValue >= 0 ? model.labelValues[0] : model.labelValues[
            new Double[] {prob, 1 - prob});
   }
}
```

3.3.3 在线更新模型

FtrlPredictStreamOp.flatMap2 函数完成了处理在线训练输出的模型数据流，在线更新模型。

LinearModelData参数是由CollectModel完成加载并且传输出来的。

在模型加载过程中，是不能预测的，没有看到相关保护机制。如果我疏漏请大家指出。

```
public void flatMap2(LinearModelData linearModel, Collector<Row> collector) throws Exce
    this.predictor.loadModel(linearModel);
}
```

0x04 问题解答

针对之前我们提出的问题，现在总结归纳如下：

- 训练阶段和预测阶段都有预制模型以应对"冷启动"嘛？都有预制模型；
- 训练阶段和预测阶段是如何关联起来的？用 linkFrom 直接把训练阶段和预测阶段的算子连在一起；
- 如何把训练出来的模型传给预测阶段？训练阶段用 Flink collector.collect 把模型发给下游算子；

- 输出模型时候，模型过大怎么处理？在线训练会 **模型打散** 之后分布发送给下游算子；
- 在线训练的模型通过什么机制实现更新？是定时驱动更新嘛？定时更新；
- 预测阶段加载模型过程中，还可以预测嘛？有没有机制保证这段时间内也能预测？目前没有发现类似保护机制；
- 训练阶段中，有哪些阶段用到了并行处理？训练过程中主要是FTRL算法的"预测predict" 和"更新参数"两个部分，以及发送模型；
- 预测阶段中，有哪些阶段用到了并行处理？预测过程中主要是分布式接受模型和分布式预测；
- 遇到高维向量如何处理？切分开嘛？切分处理；

0xFF 参考

【机器学习】逻辑回归（非常详细）

逻辑回归(logistics regression)

【机器学习】LR的分布式（并行化）实现

并行逻辑回归

机器学习算法及其并行化讨论

Online LR—— FTRL 算法理解

在线优化算法 FTRL 的原理与实现

LR+FTRL算法原理以及工程化实现

Flink流处理之迭代API分析

FTRL公式推导

FTRL论文笔记

在线机器学习FTRL(Follow-the-regularized-Leader)算法介绍

FTRL代码实现

FTRL实战之LR+FTRL（代码采用的稠密数据）

在线学习算法FTRL-Proximal原理

基于FTRL的在线CTR预测算法

CTR预测算法之FTRL-Proximal

各大公司广泛使用的在线学习算法FTRL详解

在线最优化求解(Online Optimization)之五：FTRL

FOLLOW THE REGULARIZED LEADER (FTRL) 算法总结

阅读原文