Alink漫谈(七)：如何划分训练数据集和测试数据集

目录

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。本文将为大家展现Alink如何划分训练数据集和测试数据集。

0x01 训练数据集和测试数据集

两分法

一般做预测分析时，会将数据分为两大部分。一部分是训练数据，用于构建模型，一部分是测试数据，用于检验模型。

三分法

但有时候模型的构建过程中也需要检验模型/辅助模型构建，这时会将训练数据再分为两个部分：1）训练数据；2）验证数据（Validation Data）。所以这种情况下会把数据分为三部分。

- 训练数据（Train Data）：用于模型构建。
- 验证数据（Validation Data）：可选，用于辅助模型构建，可以重复使用。
- 测试数据（Test Data）：用于检测模型构建，此数据只在模型检验时使用，用于评估模型的准确率。绝对不允许用于模型构建过程，否则会导致过渡拟合。

Training set是用来训练模型或确定模型参数的，如ANN中权值等；

Validation set是用来做模型选择（model selection），即做模型的最终优化及确定，如ANN的结构；

Test set则纯粹是为了测试已经训练好的模型的推广能力。当然test set并不能保证模型的正确性，他只是说相似的数据用此模型会得出相似的结果。

实际应用

实际应用中，一般只将数据集分成两类，即training set 和test set，大多数文章并不涉及validation set。我们这里也不涉及。大家常用的sklearn的train_test_split函数就是将矩阵随机划分为训练子集和测试子集，并返回划分好的训练集测试集样本和训练集测试集标签。

0x02 Alink示例代码

首先我们给出示例代码，然后会深入剖析：

```
public class SplitExample {
  public static void main(String[] args) throws Exception {
    String url = "iris.csv";
    String schema = "sepal_length double, sepal_width double, petal_length double, petal_width
double, category string";

    //这里是批处理
    BatchOperator data = new CsvSourceBatchOp().setFilePath(url).setSchemaStr(schema);
    SplitBatchOp spliter = new SplitBatchOp().setFraction(0.8);
    spliter.linkFrom(data);
    BatchOperator trainData = spliter;
    BatchOperator testData = spliter.getSideOutput(0);

    // 这里是流处理
    CsvSourceStreamOp dataS = new CsvSourceStreamOp().setFilePath(url).setSchemaStr(schema);
    SplitStreamOp spliterS = new SplitStreamOp().setFraction(0.4);
    spliterS.linkFrom(dataS);
    StreamOperator train_data = spliterS;
    StreamOperator test_data = spliterS.getSideOutput(0);
  }
}
```

0x03 批处理

SplitBatchOp是分割批处理的主要类，具体构建DAG的工作是在其linkFrom完成的。

总体思路比较简单：

1. 假定有一个采样比例 fraction
2. 将数据集分区，并行计算每个分区上的记录数
3. 把每个分区上的记录数累积，得到所有记录总数 totCount
4. 从上而下计算出一个采样总数：`numTarget = totCount * fraction`
5. 因为具体选择元素是在每个分区上做的，所以在每个分区上，分别计算出来这个分区应该采样的记录数，比如第n个分区上应采样记录数：`task_n_count * fraction`
6. 把这些分区 "应该采样的记录数" 累积，得出来从下而上计算出的采样总数：
   `totSelect = task_1_count * fraction + task_2_count * fraction + ... task_n_count * fraction`
7. numTarget 和 totSelect 可能不相等，所以随机决定把多出来的 `numTarget - totSelect` 加入到某一个task中。
8. 在每个task上采样得到具体的记录。

3.1 得到记录数

如果要分割数据，首先必须知道数据集的记录数。比如这个DataSet的记录是1万个？还是十万个？因为数据集可能会很大，所以这一步操作也使用了并行处理，即把数据分区，然后通过mapPartition操作得到每一个分区上元素的数目。

```
DataSet<Tuple2<Integer, Long>> countsPerPartition = DataSetUtils.countElementsPerPartition(rows
); //返回哪个task有哪些记录数

DataSet<long[]> numPickedPerPartition = countsPerPartition
    .mapPartition(new CountInPartition(fraction)) //计算总数
    .setParallelism(1)
    .name("decide_count_of_each_partition");
```

因为每个分区就对应了一个task，所以我们也可以认为，这是获取了每个task的记录数。

具体工作是在 DataSetUtils.countElementsPerPartition 中完成的。返回类型是<index of this subtask, record count in this subtask>，比如3号task拥有30个记录。

```
public static <T> DataSet<Tuple2<Integer, Long>> countElementsPerPartition(DataSet<T> input) {
   return input.mapPartition(new RichMapPartitionFunction<T, Tuple2<Integer, Long>>() {
      @Override
      public void mapPartition(Iterable<T> values, Collector<Tuple2<Integer, Long>> out) throws
 Exception {
         long counter = 0;
         for (T value : values) {
            counter++; //计算本task的记录总数
         }
         out.collect(new Tuple2<>(getRuntimeContext().getIndexOfThisSubtask(), counter));
      }
   });
}
```

计算总数的工作其实是在下一阶段算子中完成的。

3.2 随机选取记录

接下来的工作主要是在 CountInPartition.mapPartition 完成的，其作用是随机决定每个task选择多少个记录。

这时候就不需要并行了，所以 `.setParallelism(1)`

3.2.1 得到总记录数

得到了每个分区记录数之后，我们遍历每个task的记录数，然后累积得到总记录数 totCount（就是从上而下计算出来的总数）。

```
public void mapPartition(Iterable<Tuple2<Integer, Long>> values, Collector<long[]> out) throws
Exception {
        long totCount = 0L;
        List<Tuple2<Integer, Long>> buffer = new ArrayList<>();
        for (Tuple2<Integer, Long> value : values) { //遍历输入的所有分区记录
    totCount += value.f1; //f1是Long类型的记录数
    buffer.add(value);
        }
  ...
  //后续代码在下面分析。
}
```

3.2.2 决定每个task选择记录数

然后CountInPartition.mapPartition函数中会随机决定每个task会选择的记录数。mapPartition的参数 Iterable<Tuple2<Integer, Long>> values 就是前一阶段的结果 ： 一个元祖<task id, 每个task的记录数目>。

把这些元祖结合在一起，记录在buffer这个列表中。

```
buffer = {ArrayList@8972}  size = 4
 0 = {Tuple2@8975} "(3,38)" // 3号task，其对应的partition记录数是38个。
 1 = {Tuple2@8976} "(2,0)"
 2 = {Tuple2@8977} "(0,38)"
 3 = {Tuple2@8978} "(1,74)"
```

系统的task数目就是buffer大小。

```
int npart = buffer.size(); // num tasks
```

然后，根据”记录总数“计算出来 “随机训练数据的个数numTarget”。比如总数1万，应该随机分配20%，于是numTarget就应该是2千。这个数字以后会用到。

```
long numTarget = Math.round((totCount * fraction));
```

得到每个task的记录数目，比如是上面buffer中的 38，0，38，还是74，记录在 eachCount 中。

```
for (Tuple2<Integer, Long> value : buffer) {
    eachCount[value.f0] = value.f1;
}
```

得到每个task中随机选中的训练记录数，记录在 eachSelect 中。就是每个task目前 “记录数字 * fraction”。比如3号task记录数是38个，应该选20%，则38*20%=8个。

然后把这些task自己的“随机训练记录数”再累加起来得到 totSelect（就是从下而上计算出来的总数）。

```
long totSelect = 0L;
for (int i = 0; i < npart; i++) {
    eachSelect[i] = Math.round(Math.floor(eachCount[i] * fraction));
    totSelect += eachSelect[i];
}
```

请注意，这时候 totSelect 和 之前计算的numTarget就有具体细微出入了，就是理论上的一个数字，但是我们从上而下 计算 和 从下而上 计算，其结果可能不一样。通过下面我们可以看出来。

```
numTarget = all count * fraction

totSelect = task_1_count * fraction + task_2_count * fraction + ...
```

所以我们下一步要处理这个细微出入，就得到remain，这是"总体算出来的随机数目" numTarget 和 "从所有task选中的随机训练记录数累积" totSelect 的差。

```
if (totSelect < numTarget) {
    long remain = numTarget - totSelect;
    remain = Math.min(remain, totCount - totSelect);
```

如果刚好个数相等，则就正常分配。

```
if (remain == totCount - totSelect) {
```

如果数目不等，随机决定把"多出来的remain"加入到eachSelect数组中的随便一个记录上。

```
for (int i = 0; i < Math.min(remain, npart); i++) {
    int taskId = shuffle.get(i);
    while (eachSelect[taskId] >= eachCount[taskId]) {
          taskId = (taskId + 1) % npart;
    }
    eachSelect[taskId]++;
}
```

最后给出所有信息

```
long[] statistics = new long[npart * 2];
for (int i = 0; i < npart; i++) {
    statistics[i] = eachCount[i];
    statistics[i + npart] = eachSelect[i];
}
out.collect(statistics);

// 我们这里是4核，所以前面四项是eachCount，后面是eachSelect
statistics = {long[8]@9003}
 0 = 38 //eachCount
 1 = 38
 2 = 36
 3 = 38

 4 = 31 //eachSelect
 5 = 31
 6 = 28
 7 = 30
```

这些信息是作为广播变量存储起来的，马上下面就会用到。

```
 .withBroadcastSet(numPickedPerPartition, "counts")
```

3.2.3 每个task选择记录

CountInPartition.PickInPartition函数中会随机在每个task选择记录。

首先得到task数目 和 之前存储的广播变量（就是之前刚刚存储的）。

```
int npart = getRuntimeContext().getNumberOfParallelSubtasks();
List<long[]> bc = getRuntimeContext().getBroadcastVariable("counts");
```

分离count和select。

```
long[] eachCount = Arrays.copyOfRange(bc.get(0), 0, npart);
long[] eachSelect = Arrays.copyOfRange(bc.get(0), npart, npart * 2);
```

得到总task数目

```
int taskId = getRuntimeContext().getIndexOfThisSubtask();
```

得到自己 task 对应的 count, select

```
long count = eachCount[taskId];
long select = eachSelect[taskId];
```

添加本task对应的记录，随机洗牌打乱顺序

```
for (int i = 0; i < count; i++) {
     shuffle.add(i); //就是把count内的数字加到数组
}
Collections.shuffle(shuffle, new Random(taskId)); //洗牌打乱顺序

// suffle举例
shuffle = {ArrayList@8987}  size = 38
 0 = {Integer@8994} 17
 1 = {Integer@8995} 8
```

```
 2 = {Integer@8996} 33
 3 = {Integer@8997} 34
 4 = {Integer@8998} 20
 5 = {Integer@8999} 0
 6 = {Integer@9000} 26
 7 = {Integer@9001} 27
 8 = {Integer@9002} 23
 9 = {Integer@9003} 28
 10 = {Integer@9004} 9
 11 = {Integer@9005} 16
 12 = {Integer@9006} 13
 13 = {Integer@9007} 2
 14 = {Integer@9008} 5
 15 = {Integer@9009} 31
 16 = {Integer@9010} 15
 17 = {Integer@9011} 22
 18 = {Integer@9012} 18
 19 = {Integer@9013} 35
 20 = {Integer@9014} 36
 21 = {Integer@9015} 12
 22 = {Integer@9016} 7
 23 = {Integer@9017} 21
 24 = {Integer@9018} 14
 25 = {Integer@9019} 1
 26 = {Integer@9020} 10
 27 = {Integer@9021} 30
 28 = {Integer@9022} 29
 29 = {Integer@9023} 19
 30 = {Integer@9024} 25
 31 = {Integer@9025} 32
 32 = {Integer@9026} 37
 33 = {Integer@9027} 4
 34 = {Integer@9028} 11
 35 = {Integer@9029} 6
 36 = {Integer@9030} 3
 37 = {Integer@9031} 24
```

随机选择，把选择后的再排序回来

```
for (int i = 0; i < select; i++) {
    selected[i] = shuffle.get(i); //这时候select看起来是按照顺序选择，但是实际上suffle里面已经是乱序
}
Arrays.sort(selected); //这次再排序

// selected举例，一共30个
selected = {int[30]@8991}
 0 = 0
 1 = 1
 2 = 2
 3 = 5
 4 = 7
 5 = 8
 6 = 9
 7 = 10
 8 = 12
 9 = 13
 10 = 14
 11 = 15
 12 = 16
 13 = 17
```

```
 14 = 18
 15 = 19
 16 = 20
 17 = 21
 18 = 22
 19 = 23
 20 = 26
 21 = 27
 22 = 28
 23 = 29
 24 = 30
 25 = 31
 26 = 33
 27 = 34
 28 = 35
 29 = 36
```

发送选择的数据

```
if (numEmits < selected.length && iRow == selected[numEmits]) {
    out.collect(row);
    numEmits++;
}
```

3.3 设置训练数据集和测试数据集

output是训练数据集，SideOutput是测试数据集。因为这两个数据集在Alink内部都是Table类型，所以直接使用了SQL算子 `minusAll` 来完成分割。

```
this.setOutput(out, in.getSchema());
this.setSideOutputTables(new Table[]{in.getOutputTable().minusAll(this.getOutputTable())});
```

0x04 流处理

训练是在SplitStreamOp类完成的，其通过linkFrom完成了模型的构建。

流处理依赖SplitStream 和 SelectTransformation 这两个类来完成分割流。具体并没有建立一个物理操作，而只是影响了上游算子如何与下游算子联系，如何选择记录。

```
SplitStream <Row> splited = in.getDataStream().split(new RandomSelectorOp(getFraction()));
```

首先，用RandomSelectorOp来随机决定输出时候选择哪个流。我们可以看到，这里就是随便起了"a"， "b" 这两个名字而已。

```
class RandomSelectorOp implements OutputSelector <Row> {
   private double fraction;
   private Random random = null;
   @Override
   public Iterable <String> select(Row value) {
      if (null == random) {
         random = new Random(System.currentTimeMillis());
      }
      List <String> output = new ArrayList <String>(1);
      output.add((random.nextDouble() < fraction ? "a" : "b")); //随机选取数字分配，随意起的名字
      return output;
   }
}
```

其次，得到那两个随机生成的流。

```
DataStream <Row> partA = splited.select("a");
DataStream <Row> partB = splited.select("b");
```

最后把这两个流分别设置为output和sideOutput。

```
this.setOutput(partA, in.getSchema()); //训练集
this.setSideOutputTables(new Table[]{
DataStreamConversionUtil.toTable(getMLEnvironmentId(), partB, in.getSchema())}); //验证集
```

最后返回本身，这时候SplitStreamOp拥有两个成员变量：

this.output就是训练集。

this.sideOutPut就是验证集。

```
return this;
```

0x05 参考

[训练数据，验证数据和测试数据分析]