Alink漫谈(九)： 特征工程之特征哈希/标准化缩放

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。本文将剖析Alink “特征工程” 部分对应代码实现。

0x01 相关概念

1.1 特征工程

机器学习的特征工程是将原始的输入数据转换成特征，以便于更好的表示潜在的问题，并有助于提高预测模型准确性的过程。

找出合适的特征是很困难且耗时的工作，它需要专家知识，而应用机器学习基本也可以理解成特征工程。但是，特征工程对机器学习模型的应用有很大影响，有句俗话叫做“数据和特征决定了机器学习模型的性能上限”。

机器学习的输入特征包括几种：

- 数值特征：包括整形、浮点型等，可以有顺序意义，或者无序数据。
- 分类特征：如ID、性别等。
- 时间特征：时间序列如月份、年份、季度、日期、小时等。
- 空间特征：经纬度等，可以转换成邮编，城市等。
- 文本特征：文档，自然语言，语句等。

特征工程处理技巧大概有：

- 分箱（Binning）
- 独热编码（One-Hot Encoding）
- 特征哈希（Hashing Trick）
- 嵌套法（Embedding）
- 取对数（Log Transformation）
- 特征缩放（Scaling）
- 标准化（Normalization）
- 特征交互（Feature Interaction）

本文将为大家讲解特征缩放和特征哈希的实现。

1.2 特征缩放（Scaling）

特征缩放是一种用于标准化独立变量或数据特征范围的方法。 在数据处理中，它也称为数据标准化，并且通常在数据预处理步骤期间执行。特征缩放可以将很大范围的数据限定在指定范围内。由于原始数据的值范围变化很大，在一些机器学习算法中，如果没有标准化，目标函数将无法正常工作。 例如，大多数分类器按欧几里德距离计算两点之间的距离。 如果其中一个要素具有宽范围的值，则距离将受此特定要素的控制。 因此，应对所有特征的范围进行归一化，以使每个特征大致与最终距离成比例。

应用特征缩放的另一个原因是梯度下降与特征缩放比没有它时收敛得快得多。特征缩放主要包括两种：

- 最大最小缩放（Min-max Scaling）
- 标准化缩放（Standard(Z) Scaling）

1.3 特征哈希（Hashing Trick）

大多数机器学习算法的输入要求都是实数矩阵，将原始数据转换成实数矩阵就是所谓的特征工程，而特征哈希（feature hashing，也称哈希技巧，hashing trick）就是一种特征工程技术。

特征哈希的目标就是将一个数据点转换成一个向量 或者 把原始的高维特征向量压缩成较低维特征向量，且尽量不损失原始特征的表达能力。

特征哈希利用的是哈希函数将原始数据转换成指定范围内的散列值，相比较独热模型具有很多优点，如支持在线学习，维度减小很多。

比如我们将梁山好汉进行特征哈希，以关胜为例：

```
姓 名：关胜
排 名：坐第5把交椅
籍 贯：运城（今山西省-运城市）
绰 号：大刀
武 器：青龙偃月刀
星 号：天勇星
相 貌：堂堂八尺五六身躯，细细三柳髭髯，两眉入鬓，凤眼朝天，面如重枣，唇若涂朱。
原 型：南宋初，刘豫任济南知府，金军攻济南。刘豫受金人利诱，杀守将关胜，降金。这段故事被清陈忱加以演义，写入了《水浒后传》。此关胜可能就是小说中的原型。
出场回合：第063回
后 代：关铃，在《说岳全传》出场，岳云的义弟。
```

上面都是原始的输入数据，包括数值特征，分类特征，文本特征等等，计算机无法识别，必须用特征哈希转换成计算机可以识别的数据。

转换之后如下（虚构，只是展示使用 ^_^）：

```
// 假设结果是一个 30000 大小的稀疏向量，下面格式是: "index":"value"

"725":"0.8223484445229384" //姓 名
"1000":"0.8444219609970856" //排 名
"4995":"-0.18307661612028242 " //籍 贯
"8049":"0.6015161610215377" //绰 号
"8517":"-0.7340742756048447 " //武 器
"26798":":-0.734299689415312" //星 号
"24390":"0.545435" //相 貌
"25083":"0.4543543" //原 型
"25435":"-0.243432" //出场回合
"25721":"-0.7340742756048447" //后 代
```

这样关胜就变成了一个可以被程序处理的向量。

0x02 数据集

我们的数据集和示例代码都是从FTRLExample获取到的。

首先看数据集。

```
String schemaStr
                = "id string, click string, dt string, C1 string, banner_pos int, site_id strin
g, site_domain string, "
                + "site_category string, app_id string, app_domain string, app_category string,
 device_id string, "
                + "device_ip string, device_model string, device_type string, device_conn_type
string, C14 int, C15 int, "
                + "C16 int, C17 int, C18 int, C19 int, C20 int, C21 int";

//打印出前面几列看看
trainBatchData.firstN(5).print();

id|click|dt|C1|banner_pos|site_id|site_domain|site_category|app_id|app_domain|app_category|devi
ce_id|device_ip|device_model|device_type|device_conn_type|C14|C15|C16|C17|C18|C19|C20|C21
--|-----|--|--|----------|-------|-----------|-------------|------|----------|------------|----
-----|---------|------------|-----------|----------------|---|---|---|---|---|---|---|---
3199889859719711212|0|14102101|1005|0|1fbe01fe|f3845767|28905ebd|ecad2386|7801e8d9|07d7df22|a99
f214a|cfa82746|c6263d8a|1|0|15708|320|50|1722|0|35|-1|79
3200127078337687811|0|14102101|1005|1|e5c60a05|7256c623|f028772b|ecad2386|7801e8d9|07d7df22|a99
f214a|ffb0e59a|83ca6fdb|1|0|19771|320|50|2227|0|687|100075|48
3200382705425230287|1|14102101|1005|0|85f751fd|c4e18dd6|50e219e0|98fed791|d9b5648e|0f2161f8|a99
f214a|f69683cc|f51246a7|1|0|20984|320|50|2371|0|551|-1|46
3200736581912908l6|0|14102101|1005|0|1fbe01fe|f3845767|28905ebd|ecad2386|7801e8d9|07d7df22|a99f
214a|8e5b1a31|711ee120|1|0|15706|320|50|1722|0|35|100083|79
3200823995473818776|0|14102101|1005|0|f282ab5a|61eb5bc4|f028772b|ecad2386|7801e8d9|07d7df22|a99
f214a|9cf693b4|8a4875bd|1|0|18993|320|50|2161|0|35|-1|157
```

0x03 示例代码

从示例代码可以看到，首先做特征缩放，然后做特征哈希。

```
String[] selectedColNames = new String[]{
                "C1", "banner_pos", "site_category", "app_domain",
```

```
                "app_category", "device_type", "device_conn_type",
                "C14", "C15", "C16", "C17", "C18", "C19", "C20", "C21",
                "site_id", "site_domain", "device_id", "device_model"};
String[] categoryColNames = new String[]{
                "C1", "banner_pos", "site_category", "app_domain",
                "app_category", "device_type", "device_conn_type",
                "site_id", "site_domain", "device_id", "device_model"};
String[] numericalColNames = new String[]{
                "C14", "C15", "C16", "C17", "C18", "C19", "C20", "C21"};

// setup feature engineering pipeline
Pipeline featurePipeline = new Pipeline()
        .add(   // 特征缩放
                new StandardScaler()
                        .setSelectedCols(numericalColNames) // 对Double类型的列做变换
        )
        .add(   // 特征哈希
                new FeatureHasher()
                        .setSelectedCols(selectedColNames)
                        .setCategoricalCols(categoryColNames)
                        .setOutputCol(vecColName)
                        .setNumFeatures(numHashFeatures)
        );
// fit feature pipeline model
PipelineModel featurePipelineModel = featurePipeline.fit(trainBatchData);
```

0x04 标准化缩放 StandardScaler

StandardScaler的作用是把数据集的每一个特征进行标准差（standard deviation）转换 和/或 零均值化（zero mean）。transforms a dataset, normalizing each feature to have unit standard deviation and/or zero mean.

对于做特征缩放的好处，网上文章说的挺好：

> 当x全为正或者全为负时，每次返回的梯度都只会沿着一个方向发生变化，即梯度变化的方向就会向图中红色箭头所示，一会向上太多，一会向下太多。这样就会使得权重收敛效率很低。
>
> 但当x正负数量"差不多"时，就能对梯度变化方向进行"修正"，加速了权重的收敛。

让我们想想如果做标准化缩放，具体需要怎么做：

- 需要把需要处理的列的means, stdEnv这样的都计算出来，这就需要遍历整个表。所以这个是训练过程。
- 需要根据上述训练出来的means, stdEnv等数值，遍历整个表中的每个数据，应用means, stdEnv结果逐一计算。所以这个是mapper过程。

4.1 StandardScalerTrainBatchOp

StandardScalerTrainBatchOp 类做了标准化缩放相关工作。这里只对数字类型的列做转换。

```
/* StandardScaler transforms a dataset, normalizing each feature to have unit standard deviatio
n and/or zero mean. */
public class StandardScalerTrainBatchOp extends BatchOperator<StandardScalerTrainBatchOp>
    implements StandardTrainParams<StandardScalerTrainBatchOp> {

    @Override
    public StandardScalerTrainBatchOp linkFrom(BatchOperator<?>... inputs) {
        BatchOperator<?> in = checkAndGetFirst(inputs);
        String[] selectedColNames = getSelectedCols();
```

```
        StandardScalerModelDataConverter converter = new StandardScalerModelDataConverter();
        converter.selectedColNames = selectedColNames;
        converter.selectedColTypes = new TypeInformation[selectedColNames.length];

        // 获取需要转换的列
        for (int i = 0; i < selectedColNames.length; i++) {
            converter.selectedColTypes[i] = Types.DOUBLE;
        }

//得到变量如下
converter = {StandardScalerModelDataConverter@9229}
 selectedColNames = {String[8]@9228}
  0 = "C14"
  1 = "C15"
  2 = "C16"
  3 = "C17"
  4 = "C18"
  5 = "C19"
  6 = "C20"
  7 = "C21"
 selectedColTypes = {TypeInformation[8]@9231}
  0 = {FractionalTypeInfo@9269} "Double"
  1 = {FractionalTypeInfo@9269} "Double"
  2 = {FractionalTypeInfo@9269} "Double"
  3 = {FractionalTypeInfo@9269} "Double"
  4 = {FractionalTypeInfo@9269} "Double"
  5 = {FractionalTypeInfo@9269} "Double"
  6 = {FractionalTypeInfo@9269} "Double"
  7 = {FractionalTypeInfo@9269} "Double"

        // 用获取到的列信息通过 StatisticsHelper.summary 做总结，然后通过 BuildStandardScalerModel 进
行操作
        DataSet<Row> rows = StatisticsHelper.summary(in, selectedColNames)
            .flatMap(new BuildStandardScalerModel(converter.selectedColNames,
                converter.selectedColTypes,
                getWithMean(),
                getWithStd()));

        this.setOutput(rows, converter.getModelSchema());

        return this;
}
```

这里调用一环套一环，所以先打印出构建执行计划时候的调用栈给大家看看。

```
summarizer:277, StatisticsHelper (com.alibaba.alink.operator.common.statistics)
summarizer:240, StatisticsHelper (com.alibaba.alink.operator.common.statistics)
summary:71, StatisticsHelper (com.alibaba.alink.operator.common.statistics)
linkFrom:49, StandardScalerTrainBatchOp (com.alibaba.alink.operator.batch.dataproc)
train:22, StandardScaler (com.alibaba.alink.pipeline.dataproc)
fit:34, Trainer (com.alibaba.alink.pipeline)
fit:117, Pipeline (com.alibaba.alink.pipeline)
main:59, FTRLExample (com.alibaba.alink)
```

StandardScalerTrainBatchOp.linkFrom 构建出来的执行计划从逻辑上讲是：

- 1)获取需要转换的列信息

- 2)用获取到的列信息通过 StatisticsHelper.summary (StatisticsHelper类是batch statistical calculation的工具类) 做总结
  - 2.1)用 summarizer 获取table统计信息
    - 2.1.1)用in = in.select(selectedColNames);获取输入数据中需要调整的列 所对应的数据
    - 2.1.2)调用同名 summarizer 函数对in做操作进行统计
      - 2.1.2.1)调用 TableSummarizerPartition 对每个partition数据进行统计。
        - 2.1.2.1.1) 调用 TableSummarizer.visit 对本 partition 传入的Row(就是上面in的每个数据)进行计算，得出统计数据比如 squareSum，min, max, normL1。
      - 2.1.2.2)回到 summarizer 函数，调用reduce 对所有partition得的统计数据进行汇总。
  - 2.2)对 summarizer 的结果调用 summarizer.toSummary() 进行map，得到 TableSummary，其就是一个简单统计。
- 3)对 StatisticsHelper.summary 的结果 通过flatMap(BuildStandardScalerModel) 进行生成模型 / 存储操作
  - 3.1)BuildStandardScalerModel.flatMap调用 StandardScalerModelDataConverter.save
    - 3.1.1)data.add(JsonConverter.toJson(means)); 存means
    - 3.1.2)data.add(JsonConverter.toJson(stdDevs)); 存 stdDevs

具体结合代码如下

4.2 StatisticsHelper.summary

StatisticsHelper.summary 首先调用summarizer对原始输入table做总结，对应代码 2)

```
/* table summary, selectedColNames must be set. */
public static DataSet<TableSummary> summary(BatchOperator in, String[] selectedColNames) {
    return summarizer(in, selectedColNames, false) // 将会调用代码 2.1)
        .map(new MapFunction<TableSummarizer, TableSummary>() {
            @Override
            public TableSummary map(TableSummarizer summarizer) throws Exception {
                return summarizer.toSummary(); // 对应代码 2.2)
            }
        }).name("toSummary");
}
```

`summarizer(in, selectedColNames, false)` 从原始输入中获取到那些选中的列，然后继续调用另外同名函数summarizer。

```
/**
 * table stat
 */
private static DataSet<TableSummarizer> summarizer(BatchOperator in, String[]  selectedColNames
, boolean calculateOuterProduct) { // 对应代码 2.1)
    in = in.select(selectedColNames); // 对应代码2.1.1)
    return summarizer(in.getDataSet(), calculateOuterProduct, getNumericalColIndices(in.getColT
ypes()), selectedColNames); //对应代码2.1.2)
}
```

同名函数summarizer调用 TableSummarizerPartition 对每个partition处理，当然大家知道现在只是把执行计划搭建起来，不是真正的执行。当对每个partition处理完成之后，会回到这里的reduce函数进行merge。

```
/* given data, return summary. numberIndices is the indices of cols which are number type in se
lected cols. */
private static DataSet<TableSummarizer> summarizer(DataSet<Row> data, boolean bCov, int[] numbe
rIndices, String[] selectedColNames) {
    return data // mapPartition 对应代码 2.1.2.1)
        .mapPartition(new TableSummarizerPartition(bCov, numberIndices, selectedColNames))
```

```
        .reduce(new ReduceFunction<TableSummarizer>() { // reduce对应代码 2.1.2.2)
            @Override
            public TableSummarizer reduce(TableSummarizer left, TableSummarizer right) {
                return TableSummarizer.merge(left, right); //最终会merge所有的partition处理结果
            }
        });
}
```

TableSummarizerPartition针对每个partition，让每个worker用来TableSummarizer.visit来做table summary，以后会合并。对应代码 2.1.2.1.1) 。

```
/* It is table summary partition of one worker, will merge result later. */
public static class TableSummarizerPartition implements MapPartitionFunction<Row, TableSummarizer> {
    @Override
    public void mapPartition(Iterable<Row> iterable, Collector<TableSummarizer> collector) {
        TableSummarizer srt = new TableSummarizer(selectedColNames, numericalIndices, outerProduct);
        srt.colNames = selectedColNames;
        for (Row sv : iterable) {
            srt = (TableSummarizer) srt.visit(sv);
        }
        collector.collect(srt);
    }
}

// 变量如下
srt = {TableSummarizer@10742} "count: 0\n"
sv = {Row@10764} "15708,320,50,1722,0,35,-1,79"
srt.colNames = {String[8]@10733}
 0 = "C14"
 1 = "C15"
 2 = "C16"
 3 = "C17"
 4 = "C18"
 5 = "C19"
 6 = "C20"
 7 = "C21"
```

我们可以看到，上面代码中会对 iterable 做循环调用 TableSummarizer.visit函数。即通过 `visit` 来对输入的每个item（这个item就是srt.colNames对应的那些列集合起来做了一个Row）做累积计算，算出比如squareSum，min, max, normL1等等，具体在下面的变量中有体现。

```
this = {TableSummarizer@10742} "count: 1\nsum: 15708.0 320.0 50.0 1722.0 0.0 35.0 -1.0 79.0\nsquareSum: 2.46741264E8 102400.0 2500.0 2965284.0 0.0 1225.0 1.0 6241.0\nmin: 15708.0 320.0 50.0 1722.0 0.0 35.0 -1.0 79.0\nmax: 15708.0 320.0 50.0 1722.0 0.0 35.0 -1.0 79.0"
 colNames = {String[8]@10733}
 xSum = null
 xSquareSum = null
 xyCount = null
 numericalColIndices = {int[8]@10734}
 numMissingValue = {DenseVector@10791} "0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0"
 sum = {DenseVector@10792} "15708.0 320.0 50.0 1722.0 0.0 35.0 -1.0 79.0"
 squareSum = {DenseVector@10793} "2.46741264E8 102400.0 2500.0 2965284.0 0.0 1225.0 1.0 6241.0"
 min = {DenseVector@10794} "15708.0 320.0 50.0 1722.0 0.0 35.0 -1.0 79.0"
 max = {DenseVector@10795} "15708.0 320.0 50.0 1722.0 0.0 35.0 -1.0 79.0"
 normL1 = {DenseVector@10796} "15708.0 320.0 50.0 1722.0 0.0 35.0 1.0 79.0"
 vals = {Double[8]@10797}
 outerProduct = null
```

```
 count = 1
 calculateOuterProduct = false
```

4.3 BuildStandardScalerModel

这里的功能是生成模型 / 存储。

```
/* table summary build model. */
public static class BuildStandardScalerModel implements FlatMapFunction<TableSummary, Row> {
    private String[] selectedColNames;
    private TypeInformation[] selectedColTypes;
    private boolean withMean;
    private boolean withStdDevs;

    @Override
    public void flatMap(TableSummary srt, Collector<Row> collector) throws Exception {
        if (null != srt) {
            StandardScalerModelDataConverter converter = new StandardScalerModelDataConverter()
;
            converter.selectedColNames = selectedColNames;
            converter.selectedColTypes = selectedColTypes;
            // 业务
            converter.save(new Tuple3<>(this.withMean, this.withStdDevs, srt), collector);
        }
    }
}
```

save函数调用的是StandardScalerModelDataConverter.save，逻辑比较清晰：

1. 存储mean
2. 存储stdDevs
3. 构建元数据Params
4. 序列化
5. 发送序列化结果

```
/*
 * Serialize the model data to "Tuple3<Params, List<String>, List<Row>>".
 *
 * @param modelData The model data to serialize.
 * @return The serialization result.
 */
@Override
public Tuple3<Params, Iterable<String>, Iterable<Row>> serializeModel(Tuple3<Boolean, Boolean,
TableSummary> modelData) {
    Boolean withMean = modelData.f0;
    Boolean withStandarDeviation = modelData.f1;
    TableSummary summary = modelData.f2;

    String[] colNames = summary.getColNames();
    double[] means = new double[colNames.length];
    double[] stdDevs = new double[colNames.length];

    for (int i = 0; i < colNames.length; i++) {
        means[i] = summary.mean(colNames[i]); // 1. 存储mean
        stdDevs[i] = summary.standardDeviation(colNames[i]); // 2. 存储stdDevs
    }

    for (int i = 0; i < colNames.length; i++) {
        if (!withMean) {
```

```
            means[i] = 0;
        }
        if (!withStandarDeviation) {
            stdDevs[i] = 1;
        }
    }

    // 3. 构建元数据Params
    Params meta = new Params()
        .set(StandardTrainParams.WITH_MEAN, withMean)
        .set(StandardTrainParams.WITH_STD, withStandarDeviation);

    // 4. 序列化
    List<String> data = new ArrayList<>();
    data.add(JsonConverter.toJson(means));
    data.add(JsonConverter.toJson(stdDevs));

    return new Tuple3<>(meta, data, new ArrayList<>());
}
```

调用栈和变量如下，我们可以看出来模型是如何构建的。

```
save:68, RichModelDataConverter (com.alibaba.alink.common.model)
flatMap:84, StandardScalerTrainBatchOp$BuildStandardScalerModel (com.alibaba.alink.operator.bat
ch.dataproc)
flatMap:63, StandardScalerTrainBatchOp$BuildStandardScalerModel (com.alibaba.alink.operator.bat
ch.dataproc)
collect:80, ChainedFlatMapDriver (org.apache.flink.runtime.operators.chaining)
collect:35, CountingCollector (org.apache.flink.runtime.operators.util.metrics)
collect:79, ChainedMapDriver (org.apache.flink.runtime.operators.chaining)
collect:35, CountingCollector (org.apache.flink.runtime.operators.util.metrics)
run:152, AllReduceDriver (org.apache.flink.runtime.operators)
run:504, BatchTask (org.apache.flink.runtime.operators)
invoke:369, BatchTask (org.apache.flink.runtime.operators)
doRun:707, Task (org.apache.flink.runtime.taskmanager)
run:532, Task (org.apache.flink.runtime.taskmanager)
run:748, Thread (java.lang)

// 以下是输入
modelData = {Tuple3@10723}
 f0 = {Boolean@10726} true
 f1 = {Boolean@10726} true
 f2 = {TableSummary@10707} "colName|count|numMissingValue|numValidValue|sum|mean|variance|stand
ardDeviation|min|max|normL1|normL2\r\n-------|-----|---------------|-------------|---|----|----
----|----------------|---|---|------|------\nC14|399999|0.0000|399999.0000|7257042877.0000|181
42.6525|10993280.1107|3315.6116|375.0000|21705.0000|7257042877.0000|11664445.8724\nC15|399999|0
.0000|399999.0000|127629988.0000|319.0758|411.3345|20.2814|120.0000|1024.0000|127629988.0000|20
2208.2328\nC16|399999|0.0000|399999.0000|22663266.0000|56.6583|1322.7015|36.3690|20.0000|768.00
00|22663266.0000|42580.9842\nC17|399999|0.0000|399999.0000|809923879.0000|2024.8148|170166.5008
|412.5124|112.0000|2497.0000|809923879.0000|1306909.3634\nC18|399999|0.0000|399999.0000|414396.
0000|1.0360|1.5871|1.2598|0.0000|3.0000|414396.0000|1031.5736\nC19|399999|0.0000|399999.0000|77
641159.0000|194.1034|73786.4929|271.6367|33.0000|1835.0000|77641159.0000|211151.2756\nC20|39999
9|0.0000|399999.0000|16665597769.0000|41664.0986|2434589745.2799|49341.5620|-1.0000|100"
  colNames = {String[8]@10728}
  numMissingValue = {DenseVector@10729} "0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0"
  sum = {DenseVector@10730} "7.257042877E9 1.27629988E8 2.2663266E7 8.09923879E8 414396.0 7.764
1159E7 1.6665597769E10 3.0589982E7"
  squareSum = {DenseVector@10731} "1.36059297509295E14 4.0888169392E10 1.813140212E9 1.70801208
4269E12 1064144.0 4.4584861175E10 1.668188137320503E15 3.044124336E9"
  min = {DenseVector@10732} "375.0 120.0 20.0 112.0 0.0 33.0 -1.0 13.0"
```

```
  max = {DenseVector@10733} "21705.0 1024.0 768.0 2497.0 3.0 1835.0 100248.0 195.0"
  normL1 = {DenseVector@10734} "7.257042877E9 1.27629988E8 2.2663266E7 8.09923879E8 414396.0 7.
7641159E7 1.6666064771E10 3.0589982E7"
  numericalColIndices = {int[8]@10735}
  count = 399999

// 这是输出
model = {Tuple3@10816} "(Params {withMean=true, withStd=true},[[18142.652549131373,319.07576768
941925,56.658306645766615,2024.814759536899,1.035992589981475,194.1033827584569,41664.098582746
454,76.47514618786548], [3315.6115741652725,20.281383913437733,36.36896282478844,412.5124249687
0356,1.259797591740416,271.6366927754722,49341.56204742555,41.9748291967459651],[])"
 f0 = {Params@10817} "Params {withMean=true, withStd=true}"
 f1 = {ArrayList@10820}  size = 2
  0 = "[18142.652549131373,319.07576768941925,56.658306645766615,2024.814759536899,1.0359925899
81475,194.1033827584569,41664.098582746454,76.47514618786548]"
  1 = "[3315.6115741652725,20.281383913437733,36.36896282478844,412.51242496870356,1.2597975917
40416,271.6366927754722,49341.56204742555,41.974829196745965]"
 f2 = {ArrayList@10818}  size = 0
```

4.4 转换 mapper

训练好之后，当转换时候，会对每个item row进行map，这里面使用之前计算出来的 means/stdDevs 进行具体标准化。

```
@Override
public Row map(Row row) throws Exception {
    Row r = new Row(this.selectedColIndices.length);
    for (int i = 0; i < this.selectedColIndices.length; i++) {
        Object obj = row.getField(this.selectedColIndices[i]);
        if (null != obj) {
            if (this.stddevs[i] > 0) {
                double d = (((Number) obj).doubleValue() - this.means[i]) / this.stddevs[i];
                r.setField(i, d);
            } else {
                r.setField(i, 0.0);
            }
        }
    }
    return this.predResultColsHelper.getResultRow(row, r);
}

// means, stddevs 是对应那几列之前统计出来的总体数值，是根据这些来进行转换的。
this = {StandardScalerModelMapper@10909}
 selectedColNames = {String[8]@10873}
 selectedColTypes = {TypeInformation[8]@10874}
 selectedColIndices = {int[8]@10912}
 means = {double[8]@10913}
  0 = 18142.652549131373
...
  7 = 76.47514618786548
 stddevs = {double[8]@10914}
  0 = 3315.6115741652725
...
  7 = 41.974829196745965
```

变量如下，Row是输入数据，r 是对那几个需要转换的数据进行转换之后，生成的数据。

标准化之后，用 OutputColsHelper.getResultRow把 Row 和 r 归并起来。

```
row = {Row@10865} "3200382705425230287,1,14102101,1005,0,85f751fd,c4e18dd6,50e219e0,98fed791,d9b5648e,0f2161f8,a99f214a,f69683cc,f51246a7,1,0,20984,320,50,2371,0,551,-1,46"
```
其中 "20984,320,50,2371,0,551,-1,46" 是需要转换的数据。

```
r = {Row@10866} "0.8569602884149525,0.04557047559108551,-0.18307661612028242,0.8392116685682023,-0.8223484445229384,1.313874843618953,-0.8444219609970856,-0.7260338343491822"
```
这里是上面需要转换的数据进行标准化之后的结果

堆栈如下

```
getResultRow:177, OutputColsHelper (com.alibaba.alink.common.utils)
map:88, StandardScalerModelMapper (com.alibaba.alink.operator.common.dataproc)
map:43, ModelMapperAdapter (com.alibaba.alink.common.mapper)
map:18, ModelMapperAdapter (com.alibaba.alink.common.mapper)
run:103, MapDriver (org.apache.flink.runtime.operators)
run:504, BatchTask (org.apache.flink.runtime.operators)
invoke:369, BatchTask (org.apache.flink.runtime.operators)
doRun:707, Task (org.apache.flink.runtime.taskmanager)
run:532, Task (org.apache.flink.runtime.taskmanager)
run:748, Thread (java.lang)
```

0x05 特征哈希 FeatureHasher

FeatureHasher完成了特征哈希功能，这个没有训练，就是mapper。具体细节是：

- 把 categorical特征 或者 数值特征 投射到给定领域的特征向量上。
- 使用 MurMurHash3 算法。
- 对于categorical特征，使用"colName=value"进行哈希。colName是特征列名，value是特征值。相关哈希值是 1.0
- 对于数值特征，使用"colName"做哈希。相关哈希值就是特征值
- categorical特征 或者 数值特征 是自动发现的。

对应代码看。

5.1 稀疏矩阵

最终生成了一个30000大小的，最后名字是"vec"的特征矩阵。这里是稀疏矩阵。

```
String vecColName = "vec";
int numHashFeatures = 30000;
// setup feature engineering pipeline
Pipeline featurePipeline = new Pipeline()
        .add(
                new StandardScaler()
                        .setSelectedCols(numericalColNames)
        )
        .add(
                new FeatureHasher()
                        .setSelectedCols(selectedColNames)
                        .setCategoricalCols(categoryColNames)
                        .setOutputCol(vecColName)
                        .setNumFeatures(numHashFeatures)
        );
```

5.2 FeatureHasherMapper

传入map函数时候，Row就是 “原始数据经过标准化处理之后的数据”。

遍历数值特征列，进行哈希变换；遍历categorical特征列，进行哈希转换。

```
public class FeatureHasherMapper extends Mapper {
    /**
     * Projects a number of categorical or numerical features into a feature vector of a specif
ied dimension.
     *
     * @param row the input Row type data
     * @return the output row.
     */
    @Override
    public Row map(Row row) {
        TreeMap<Integer, Double> feature = new TreeMap<>();
        // 遍历数值特征列，进行哈希变换；
        for (int key : numericColIndexes) {
            if (null != row.getField(key)) {
                double value = ((Number)row.getField(key)).doubleValue();
                String colName = colNames[key];
                updateMap(colName, value, feature, numFeature);
            }
        }
        // 遍历categorical特征列，进行哈希转换
        for (int key : categoricalColIndexes) {
            if (null != row.getField(key)) {
                String colName = colNames[key];
                updateMap(colName + "=" + row.getField(key).toString(), 1.0, feature, numFeatur
e);
            }
        }

        return outputColsHelper.getResultRow(row, Row.of(new SparseVector(numFeature, feature))
);
    }
}

//运行时候打印变量如下
selectedCols = {String[19]@9817}
 0 = "C1"
 1 = "banner_pos"
 2 = "site_category"
 3 = "app_domain"
 4 = "app_category"
 5 = "device_type"
 6 = "device_conn_type"
 7 = "C14"
 8 = "C15"
 9 = "C16"
 10 = "C17"
 11 = "C18"
 12 = "C19"
 13 = "C20"
 14 = "C21"
 15 = "site_id"
 16 = "site_domain"
 17 = "device_id"
 18 = "device_model"

numericColIndexes = {int[8]@10789}
 0 = 16
 1 = 17
 2 = 18
```

```
 3 = 19
 4 = 20
 5 = 21
 6 = 22
 7 = 23

categoricalColIndexes = {int[11]@10791}
 0 = 3
 1 = 4
 2 = 7
 3 = 9
 4 = 10
 5 = 14
 6 = 15
 7 = 5
 8 = 6
 9 = 11
 10 = 13
```

5.3 哈希操作 updateMap

updateMap完成了具体哈希操作，用哈希函数生成了稀疏矩阵的index，然后把value放入对应的index中。

具体哈希函数使用 `org.apache.flink.shaded.guava18.com.google.common.hash` 。

```
/* Update the treeMap which saves the key-value pair of the final vector, use the hash value of
 the string as key
 * and the accumulate the corresponding value.
 *
 * @param s     the string to hash
 * @param value the accumulated value */
private static void updateMap(String s, double value, TreeMap<Integer, Double> feature, int num
Feature) {
    // HASH = {Murmur3_32HashFunction@10755} "Hashing.murmur3_32(0)"
    int hashValue = Math.abs(HASH.hashUnencodedChars(s).asInt());

    int index = Math.floorMod(hashValue, numFeature);
    if (feature.containsKey(index)) {
        feature.put(index, feature.get(index) + value);
    } else {
        feature.put(index, value);
    }
}
```

比如当如下输入时候，得到index是26798，所以会在vec中的 26798 中设置Value

```
s = "C14"
value = 0.33428145187593655
feature = {TreeMap@10836}  size = 1
 {Integer@10895} 26798 -> {Double@10896} 0.33428145187593655
numFeature = 30000
hashValue = 23306798
index = 26798
```

最终特征哈希之后，得到的vec会附加在原始Row上的第25项（原来是24项，现在在最后附加一项），就是下面的 `24 = {SparseVector@10932}` 。

```
row = {Row@10901}
 fields = {Object[25]@10907}
```

```
  0 = "3199898597197112 12"
  1 = "0"
  2 = "14102101"
  3 = "1005"
  4 = {Integer@10912} 0
  5 = "1fbe01fe" // "device_type" 是这个数值，这个是原始输入，大家如果遗忘可以回头看看示例代码输出。
  6 = "f3845767"
  7 = "28905ebd"
  8 = "ecad2386"
  9 = "7801e8d9"
  10 = "07d7df22"
  11 = "a99f214a"
  12 = "cfa82746"
  13 = "c6263d8a"
  14 = "1"
  15 = "0"
  16 = {Double@10924} -0.734299689415312
  17 = {Double@10925} 0.04557047559108551
  18 = {Double@10926} -0.18307661612028242
  19 = {Double@10927} -0.7340742756048447
  20 = {Double@10928} -0.8223484445229384
  21 = {Double@10929} -0.5857212482334542
  22 = {Double@10930} -0.8444219609970856
  23 = {Double@10931} 0.060151616110215377
  24 = {SparseVector@10932} "$30000$725:-0.8223484445229384 1000:1.0 3044:-0.8444219609970856 4
995:-0.18307661612028242 8049:0.060151616110215377 8517:1.0 10962:1.0 17954:1.0 18556:1.0 21430
:1.0 23250:1.0 24010:1.0 24390:1.0 25083:0.04557047559108551 25435:-0.5857212482334542 25721:-0
.7340742756048447 26169:1.0 26798:-0.734299689415312 29671:1.0"

// 30000 表示一共是30000大小的稀疏向量
// 725:-0.8223484445229384 表示第725的item中的数值是-0.8223484445229384，依次类推。
```

0xFF 参考

[深度学习图像预处理中为什么使用零均值化(zero-mean)]

[数据特征处理之特征哈希（Feature Hashing）]

[特征工程相关技术简介]

[特征哈希（Feature Hashing）]