Alink漫谈(四)：模型的来龙去脉

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。本文将从模型角度入手带领大家来再次深入Alink。

因为Alink的公开资料太少，所以以下均为自行揣测，肯定会有疏漏错误，希望大家指出，我会随时更新。

0x01 模型

之前的文章中，我们一直没有仔细说明Alink的模型，本篇我们就深入探究一下。套用下范伟的话：我既想知道模型是怎么来的,我又想知道模型是怎么没的。

1.1 模型包含内容

我们先想想，一个机器学习训练出来的模型，应该包含哪些内容。

- 流水线：因为一个模型可能包括多个阶段，比如转化，预测等，这样构成了一个流水线。
- 算法 ： 这个是具体机器学习平台绑定的。比如在Flink就是某一个java算法类。
- 参数：这个是肯定要有的，机器学习很大一部分工作不就是做这个的嘛。
- 数据：这个其实也应该算参数的一种，也是训练出来的。比如说KMeans算法训练出来的各个中心点。

1.2 Alink的模型文件

让我们打开Alink的模型文件来验证下：

```
-1,"{""schema"":[""""],""model_id BIGINT,model_info VARCHAR""],""param"":[""{\""outputCol\"":\""
\\\""features\\\""\"",\""selectedCols\"":\""[\\\""sepal_length\\\"",\\\""sepal_width\\\"",\\\""
petal_length\\\"",\\\""petal_width\\\""]\""}"",""{\""vectorCol\"":\""\\\""features\\\""\"",\""m
axIter\"":\""100\"",\""reservedCols\"":\""[\\\""category\\\""]\"",\""k\"":\""3\"",\""prediction
Col\"":\""\\\""prediction_result\\\""\"",\""predictionDetailCol\"":\""\\\""prediction_detail\\\
""\""}""],""clazz"":[""com.alibaba.alink.pipeline.dataproc.vector.VectorAssembler"",""com.aliba
ba.alink.pipeline.clustering.KMeansModel""]}"

1,"0^{""vectorCol"":""\""features\"""",""latitudeCol"":null,""longitudeCol"":null,""distanceTyp
e"":""\""EUCLIDEAN\"""",""k"":""3"",""vectorSize"":""4""}"
1,"1048576^{""clusterId"":0,""weight"":39.0,""vec"":{""data"":[6.853846153846153,3.07692307692
30766,5.7153846153846155,2.0538461538461545]}}"
1,"2097152^{""clusterId"":1,""weight"":61.0,""vec"":{""data"":[5.883606557377049,2.740983606557
377,4.388524590163936,1.4344262295081969]}}"
1,"3145728^{""clusterId"":2,""weight"":50.0,""vec"":{""data"":[5.006,3.418,1.4640000000000002,0
.24400000000000005]}}"
```

我们看到了两个类名字：

com.alibaba.alink.pipeline.dataproc.vector.VectorAssembler

com.alibaba.alink.pipeline.clustering.KMeansModel

这就是我们提到的算法，Alink在执行过程中，可以根据这两个类名字来生成java类。而两个算法类看起来是可以构建成一个流水线。我们也能看到参数和数据。

但是有几个地方很奇怪：

- 1048576，2097152这些奇怪的数字是什么意思？
- 为什么文件的第一个数值是-1？然后第二行第一个数字是 1？怎么没有 中间的 0 ？
- 具体Alink是如何生成和加载模型的？

下面我们就一一排查。

0x02 流程图

我们首先给出一个流程图便于大家理解。这个图只是逻辑上的大致概念，和真实运行有区别。因为实际场景上是先生成执行计划，再具体操作。

```
* 下面只是逻辑上的大致概念，和真实运行有区别，因为实际场景上是先生成执行计划，再具体操作.
* 所以只是给大家一个概念。
*
*
*     Pipeline.fit 训练
*         |
*         |
*         +-----> KMeansTrainModelData [ centroids, params -- 中心点数据，参数]
*         |       // KMeansOutputModel.calc()中执行，生成中心点数据和参数
*         |
*         |
*         +-----> Tuple2<Params, Iterable<String>> [ "Params"是模型元数据，Iterable<String>是模型具
体数据 ]
*         |       // KMeansModelDataConverter.serializeModel()，进行序列化操作，包括 把数据转换成json,
调用KMeansTrainModelData.toParams设置各种参数
*         |
*         |
*         +-----> Collector<Row> [ Row可以有任意的field，基于position(zero-based)访问field ]
*         |       // ModelConverterUtils.appendMetaRow, ModelConverterUtils.appendDataRows
*         |
*         |
```

```
*         +-----> List<Row> model [ collector.getRows() ]
*         |       // List<Row> model = completeResult.calc(context);
*         |
*         |
*         +-----> DataSet<Row> [ 序列化算子计算结果 ]
*         |       // BaseComQueue.exec --- serializeModel(clearObjs(loopEnd))
*         |
*         |
*         +-----> Table output [ AlgoOperator.output, 就是算子组件的输出表 ]
*         |       // KMeansTrainBatchOp.linkFrom --- setOutput
*         |
*         |
*         +-----> KMeansModel [ 模型, Find the closest cluster center for every point ]
*         |       // createModel(train(input).getOutputTable()) 这里设定模型参数
*         |       // KMeansModel.setModelData(Table modelData) 这里设定模型数据
*         |
*         |
*         +-----> TransformerBase[] [ PipelineModel.transformers ]
*         |       // 这就是最终训练出来的流水线模型，KMeansModel是其中一个，KMeansModelMapper是KMeansMod
el的业务组件
*         |
*         |
*     PipelineModel.save 存储
*         |
*         |
*         +-----> BatchOperator [ 把transformers数组压缩成BatchOperator ]
*         |       // ModelExporterUtils.packTransformersArray
*         |
*         |
*         +-----> 存储的模型文件 [ csv文件 ]
*         |       // PipelineModel.save --- CsvSinkBatchOp(path)
*         |
*     PipelineModel.load 加载
*         |
*         |
*         +-----> 存储的模型文件 [ csv文件 ]
*         |       // PipelineModel.load --- CsvSourceBatchOp(path)
*         |
*         |
*         +-----> KMeansModel [ 模型, Find the closest cluster center for every point ]
*         |       // 依据文件生成模型, (TransformerBase) clazz.getConstructor(Params.class)
*         |       // 设置数据((ModelBase) transformers[i]).setModelData(data.getOutputTable())
*         |
*         +-----> TransformerBase[] [ 从csv文件读取并恢复的transformers ]
*         |       // ModelExporterUtils.unpackTransformersArray(batchOp)
*         |
*         |
*         +-----> PipelineModel [ 流水线模型 ]
*         |       // new PipelineModel(ModelExporterUtils.unpackTransformersArray(batchOp));
*         |
*         |
*     PipelineModel.transform(data) 预测
*         |
*         |
*         |
*         +-----> ModelSource [ Load model data from ModelSource when open() ]
*         |       // ModelMapperAdapter.open --- List<Row> modelRows = this.modelSource.getModel
Rows(getRuntimeContext());
*         |
*         +-----> Tuple2<Params, Iterable<String>> [ metaAndData ]
```

```
*         |       // SimpleModelDataConverter.load
*         |
*         |
*         +-----> KMeansTrainModelData [ 反序列化 ]
*         |       // KMeansModelDataConverter.deserializeModel(Params params, Iterable<String> d
ata)
*         |
*         |
*         +-----> KMeansTrainModelData [ Load KMeansTrainModelData from saved model ]
*         |       // KMeansModelMapper.loadModel
*         |       // KMeansTrainModelData.loadModelForTrain(Params params, Iterable<String> data
)
*         |
*         |
*         +-----> KMeansPredictModelData [ Model data for KMeans trainData ]
*         |       // 将训练模型数据转换为预测模型数据，里面包含centroids
*         |       // KMeansUtil.transformTrainDataToPredictData(trainModelData);
*         |
*         |
*         +-----> Row row [ "5.0,3.2,1.2,0.2,Iris-setosa,5.0 3.2 1.2 0.2" ]
*         |       // row是预测目标数据，ModelMapperAdapter.map
*         |
*         |
*         +-----> Row row [ "0|0.4472728134421832 0.35775115900088217 0.19497602755693455" ]
*         |       // 预测结果，KMeansModelMapper.map
*         |
*         |
```

0x03 生成模型

我们还是用KMeans算法来做示例，看看模型数据是什么样子，如何转换成Alink需要的样子。

```
VectorAssembler va = new VectorAssembler()
    .setSelectedCols(new String[]{"sepal_length", "sepal_width", "petal_length", "petal_width"}
)
    .setOutputCol("features");

KMeans kMeans = new KMeans().setVectorCol("features").setK(3)
    .setPredictionCol("prediction_result")
    .setPredictionDetailCol("prediction_detail")
    .setReservedCols("category")
    .setMaxIter(100);

Pipeline pipeline = new Pipeline().add(va).add(kMeans);
pipeline.fit(data);
```

从之前文章中大家可以知道，KMeans训练最重要的类是KMeansTrainBatchOp。KMeansTrainBatchOp在算法迭代结束时候，使用

```
.closeWith(new KMeansOutputModel(distanceType, vectorColName, latitudeColName,
longitudeColName))
```

来输出模型。

3.1 生成模型

所以我们重点就要看 KMeansOutputModel 类。其calc函数的作用就是把中心点和参数转化为模型。

- 首先是调用serializeModel将中心点序列化成json。这里记做 (1)，下面代码注释会对应指出。

- 其次save函数会进行序列化，生成了一个Tuple2 <Params, Iterable>。Params是参数，Iterable是模型的具体数据，就是中心点的集合。这里记做 (2)，下面代码注释会对应指出。
- 然后save函数把参数和数据分开存储。这里记做 (3)，下面注释会对应指出。
- 最后collector就是模型数据。这里记做 (4)，下面注释会对应指出。

```
/**
 * Tranform the centroids to KmeansModel.
 */
public class KMeansOutputModel extends CompleteResultFunction {
   private DistanceType distanceType;
   private String vectorColName;
   private String latitudeColName;
   private String longtitudeColName;
   @Override
         public List <Row> calc(ComContext context) {
                        KMeansTrainModelData modelData = new KMeansTrainModelData();
      ... 各种赋值操作
                        modelData.params = new KMeansTrainModelData.ParamSummary();
                        modelData.params.k = k;
                        modelData.params.vectorColName = vectorColName;
      ...

// 我们可以看出来，在此处，计算出来的中心点和各种参数已经被添加到KMeansTrainModelData之中。
modelData = {KMeansTrainModelData@11319}
 centroids = {ArrayList@11327}  size = 3
  0 = {KMeansTrainModelData$ClusterSummary@11330}
   clusterId = 0
   weight = 38.0
   vec = {DenseVector@11333} "6.849999999999999 3.0736842105263156 5.742105263157895 2.07105263
1578947"
  1 = {KMeansTrainModelData$ClusterSummary@11331}
  2 = {KMeansTrainModelData$ClusterSummary@11332}
 params = {KMeansTrainModelData$ParamSummary@11328}
  k = 3
  vectorSize = 4
  distanceType = {DistanceType@11287} "EUCLIDEAN"
  vectorColName = "features"
  latitudeColName = null
  longtitudeColName = null

                        RowCollector collector = new RowCollector();
      // save函数中将进行(1)(2)(3)，后续代码中会具体给出(1)(2)(3)的位置
                        new KMeansModelDataConverter().save(modelData, collector);

     // KMeansModelDataConverter实现了SimpleModelDataConverter，所以save就调用到了KMeansModelDataCo
nverter.save，其调用serializeModel将中心点转换jason。最后生成了一个Tuple2 <Params, Iterable<String>>

    // (4) 这时候collector就是模型数据。
                        return collector.getRows();

// 我们能看出来，模型数据已经和模型文件的内容有几分相似了。里面有之前提到的奇怪数字。
collector = {RowCollector@11321}
 rows = {ArrayList@11866}  size = 4
  0 = {Row@11737} "0,{"vectorCol":"\"features\"","latitudeCol":null,"longitudeCol":null,"distan
ceType":"\"EUCLIDEAN\"","k":"3","vectorSize":"4"}"
  1 = {Row@11801} "1048576,{"clusterId":0,"weight":38.0,"vec":{"data":[6.849999999999999,3.0736
842105263156,5.742105263157895,2.071052631578947]}}"
  2 = {Row@11868} "2097152,{"clusterId":1,"weight":50.0,"vec":{"data":[5.006,3.4179999999999997
,1.4640000000000002,0.24400000000000002]}}"
```

```
  3 = {Row@11869} "3145728,{"clusterId":2,"weight":62.0,"vec":{"data":[5.901612903225806,2.7483870967741937,4.393548387096773,1.4338709677419355]}}"
   }
}
```

具体转化是在KMeansModelDataConverter和其基类SimpleModelDataConverter中完成。首先是调用serializeModel将中心点序列化成json，形成了一个json列表。

```
/**
 * KMeans Model.
 * Save the id, center point and point number of clusters.
 */
public class KMeansModelDataConverter extends SimpleModelDataConverter<KMeansTrainModelData, KMeansPredictModelData> {
   public KMeansModelDataConverter() {}

   @Override
   public Tuple2<Params, Iterable<String>> serializeModel(KMeansTrainModelData modelData) {
      List <String> data = new ArrayList <>();
      for (ClusterSummary centroid : modelData.centroids) {
         data.add(JsonConverter.toJson(centroid));  // (1), 把中心点转换生成json
      }
      return Tuple2.of(modelData.params.toParams(), data);
   }

   @Override
   public KMeansPredictModelData deserializeModel(Params params, Iterable<String> data) {
      KMeansTrainModelData trainModelData = KMeansUtil.loadModelForTrain(params, data);
      return KMeansUtil.transformTrainDataToPredictData(trainModelData);
   }
}
```

其次进行序列化操作，生成Tuple2<Params, Iterable>。

```
/**
 * The abstract class for a kind of {@link ModelDataConverter} where the model data can serialize to
 * "Tuple2&jt;Params, Iterable&jt;String&gt;&gt;". Here "Params" is the meta data of the model, and "Iterable&jt;String&gt;" is
 * concrete data of the model.
 */
public abstract class SimpleModelDataConverter<M1, M2> implements ModelDataConverter<M1, M2> {
    @Override
    public M2 load(List<Row> rows) {
        Tuple2<Params, Iterable<String>> metaAndData = ModelConverterUtils.extractModelMetaAndData(rows);
        return deserializeModel(metaAndData.f0, metaAndData.f1);
    }

    @Override
    public void save(M1 modelData, Collector<Row> collector) {
// (2), 序列化生成Tuple2
        Tuple2<Params, Iterable<String>> model = serializeModel(modelData);

// 此时模型数据是一个元祖Tuple2<Params, Iterable<String>>
model = {Tuple2@11504} "(Params {vectorCol="features", latitudeCol=null, longitudeCol=null, distanceType="EUCLIDEAN", k=3, vectorSize=4},[{"clusterId":0,"weight":38.0,"vec":{"data":[6.849999999999999,3.0736842105263156,5.742105263157895,2.071052631578947]}}, {"clusterId":1,"weight":50.0,"vec":{"data":[5.006,3.4179999999999997,1.4640000000000002,0.24400000000000002]}}, {"cluster
```

```
Id":2,"weight":62.0,"vec":{"data":[5.901612903225806,2.7483870967741937,4.393548387096773,1.433
8709677419355]}}])"

// (3) 分开发送参数和数据
        ModelConverterUtils.appendMetaRow(model.f0, collector, 2);
        ModelConverterUtils.appendDataRows(model.f1, collector, 2);
    }
}
```

然后分开存储参数和数据。

```
/**
 * Collector of Row type data.
 */
public class RowCollector implements Collector<Row> {
  private List<Row> rows;
        @Override
        public void collect(Row row) {
                rows.add(row); // 把数据存储起来
        }
}
// 调用栈是
collect:37, RowCollector (com.alibaba.alink.common.utils)
collect:12, RowCollector (com.alibaba.alink.common.utils)
appendStringData:270, ModelConverterUtils (com.alibaba.alink.common.model)
appendMetaRow:35, ModelConverterUtils (com.alibaba.alink.common.model)
save:57, SimpleModelDataConverter (com.alibaba.alink.common.model)
calc:76, KMeansOutputModel (com.alibaba.alink.operator.common.clustering.kmeans)
mapPartition:287, BaseComQueue$4 (com.alibaba.alink.common.comqueue)
```

3.2 转换DataSet

模型数据是要转换成 DataSet，即 a collection of rows。其转换目的是为了让模型数据在Alink中更好的传输和被利用。

把模型数据中的string转换为 row数据的时候，可能会遇到string过长的问题，所以Alink就将String分割转存为多行row。这时候就用ModelConverterUtils的getModelId，getStringIndex函数来分割。

这时候得到的model Id就是计算出来的1048576，就是模型文件中的那个奇怪数字。

后续load模型时候也会用同样思路从row转换回模型string。

```
// A utility class for converting model data to a collection of rows.
class ModelConverterUtils {
    /**
     * Maximum number of slices a string can split to.
     */
    static final long MAX_NUM_SLICES = 1024L * 1024L;

    private static long getModelId(int stringIndex, int sliceIndex) {
        return MAX_NUM_SLICES * stringIndex + sliceIndex;
    }

    private static int getStringIndex(long modelId) {
        return (int) ((modelId) / MAX_NUM_SLICES);
    }
}

row = {Row@11714} "1048576,{"clusterId":0,"weight":62.0,"vec":{"data":[5.901612903225806,2.7483
870967741932,4.393548387096773,1.4338709677419355]}}"
```

```
 fields = {Object[2]@11724}
  0 = {Long@11725} 1048576
  1 = "{"clusterId":0,"weight":62.0,"vec":{"data":[5.901612903225806,2.7483870967741932,4.39354
8387096773,1.4338709677419355]}}"

// 相关调用栈如下
appendStringData:270, ModelConverterUtils (com.alibaba.alink.common.model)
appendDataRows:52, ModelConverterUtils (com.alibaba.alink.common.model)
save:58, SimpleModelDataConverter (com.alibaba.alink.common.model)
calc:76, KMeansOutputModel (com.alibaba.alink.operator.common.clustering.kmeans)
mapPartition:287, BaseComQueue$4 (com.alibaba.alink.common.comqueue)
run:103, MapPartitionDriver (org.apache.flink.runtime.operators)
...
run:748, Thread (java.lang)
```

3.3 存储为Table

前面KMeansOutputModel最终返回的是一个DataSet，这里将把这个DataSet转化为Table存储在流水线中。

```
public final class KMeansTrainBatchOp extends BatchOperator <KMeansTrainBatchOp>

        public KMeansTrainBatchOp linkFrom(BatchOperator <?>... inputs) {
                DataSet <Row> finalCentroid = iterateICQ(initCentroid, data,
                        vectorSize, maxIter, tol, distance, distanceType, vectorColName, null,
null);

    // 这里存储为Table
                this.setOutput(finalCentroid, new KMeansModelDataConverter().getModelSchema());
                return this;
}

this = {KMeansTrainBatchOp@5130} "UnnamedTable$1"
 params = {Params@5143} "Params {vectorCol="features", maxIter=100, reservedCols=["category"],
k=3, predictionCol="prediction_result", predictionDetailCol="prediction_detail"}"
 output = {TableImpl@5188} "UnnamedTable$1"
  tableEnvironment = {BatchTableEnvironmentImpl@5190}
  operationTree = {DataSetQueryOperation@5191}
  operationTreeBuilder = {OperationTreeBuilder@5192}
  lookupResolver = {LookupCallResolver@5193}
  tableName = "UnnamedTable$1"
 sideOutputs = null
```

我们可以看到，在Alink运行时候，模型数据都统一转化为Table类型。这部分原因可能是因为Alink想要统一处理DataSet和DataStream，即批和流都要用一个思路或者代码来处理。而Flink目前已经用Table来统一整合二者，所以Alink就针对此统一用Table。参见如下：

```
public abstract class ModelBase<M extends ModelBase<M>> extends TransformerBase<M>
    implements Model<M> {
    protected Table modelData;
}

public abstract class AlgoOperator<T extends AlgoOperator<T>>
    implements WithParams<T>, HasMLEnvironmentId<T>, Serializable {
    // Params for algorithms.
    private Params params;

    // The table held by operator.
    private Table output = null;
```

```
    // The side outputs of operator that be similar to the stream's side outputs.
    private Table[] sideOutputs = null;
}
```

0x04 存储模型

4.1 存储代码

我们修改一下代码，调用save函数把流水线模型存储起来。Alink目前是把模型文件存储成特殊格式的csv文件。

```
Pipeline pipeline = new Pipeline().add(va).add(kMeans);
pipeline.fit(data).save("./kmeans.csv");
```

流水线存储代码如下：

```
public class PipelineModel extends ModelBase<PipelineModel> implements LocalPredictable {
  // Pack the pipeline model to a BatchOperator.
  public BatchOperator save() {
      return ModelExporterUtils.packTransformersArray(transformers);
  }
}
```

我们可以看到，流水线最终调用到 ModelExporterUtils.packTransformersArray，所以我们就重点看看这个函数。这里可以解答模型文件中的问题：为什么第一个数值是-1？然后是 1？怎么没有 中间的 0 ？

模型文件中每行第一个数字对应的是transformer的index。config是特殊的所以index设置为-1，下面代码中有指出。

模型文件中的1 就是说明第二个transformer KMeansModel具有数据，具体数据内容就在index 1对应这行 。

为什么模型文件没有 0 就是因为第一个transformer VectorAssembler没有自己的数据，所以就不包括了。

```
class ModelExporterUtils {
    //Pack an array of transformers to a BatchOperator.
    static BatchOperator packTransformersArray(TransformerBase[] transformers) {
        int numTransformers = transformers.length;
        String[] clazzNames = new String[numTransformers];
        String[] params = new String[numTransformers];
        String[] schemas = new String[numTransformers];
        for (int i = 0; i < numTransformers; i++) {
            clazzNames[i] = transformers[i].getClass().getCanonicalName();
            params[i] = transformers[i].getParams().toJson();
            schemas[i] = "";
            if (transformers[i] instanceof PipelineModel) {
                schemas[i] = CsvUtil.schema2SchemaStr(PIPELINE_MODEL_SCHEMA);
            } else if (transformers[i] instanceof ModelBase) {
                long envId = transformers[i].getMLEnvironmentId();
                BatchOperator data = BatchOperator.fromTable(((ModelBase) transformers[i]).getM
odelData());
                data.setMLEnvironmentId(envId);
                data = data.link(new VectorSerializeBatchOp().setMLEnvironmentId(envId));
                schemas[i] = CsvUtil.schema2SchemaStr(data.getSchema());
            }
        }
        Map<String, Object> config = new HashMap<>();
        config.put("clazz", clazzNames);
        config.put("param", params);
        config.put("schema", schemas);
        // 这里就对应着模型文件的第一个数值 -1，就是config对应的index就是-1。
        Row row = Row.of(-1L, JsonConverter.toJson(config));
```

```
 // 这个时候我们可以看到，schema，param，clazz 就是对应着模型文件中的输出，我们距离目标更近了一步
 config = {HashMap@5432}  size = 3
 "schema" -> {String[2]@5431}
  key = "schema"
  value = {String[2]@5431}
   0 = ""
   1 = "model_id BIGINT,model_info VARCHAR"
 "param" -> {String[2]@5430}
  key = "param"
  value = {String[2]@5430}
   0 = "{"outputCol":"\"features\"","selectedCols":"[\"sepal_length\",\"sepal_width\",\"petal_l
ength\",\"petal_width\"]"}"
   1 = "{"vectorCol":"\"features\"","maxIter":"100","reservedCols":"[\"category\"]","k":"3","pr
edictionCol":"\"prediction_result\"","predictionDetailCol":"\"prediction_detail\""}"
 "clazz" -> {String[2]@5429}
  key = "clazz"
  value = {String[2]@5429}
   0 = "com.alibaba.alink.pipeline.dataproc.vector.VectorAssembler"
   1 = "com.alibaba.alink.pipeline.clustering.KMeansModel"

        BatchOperator packed = new MemSourceBatchOp(Collections.singletonList(row), PIPELINE_MO
DEL_SCHEMA)
            .setMLEnvironmentId(transformers.length > 0 ? transformers[0].getMLEnvironmentId()
:
                MLEnvironmentFactory.DEFAULT_ML_ENVIRONMENT_ID);
        for (int i = 0; i < numTransformers; i++) {
            BatchOperator data = null;
            final long envId = transformers[i].getMLEnvironmentId();
            if (transformers[i] instanceof PipelineModel) {
                data = packTransformersArray(((PipelineModel) transformers[i]).transformers);
            } else if (transformers[i] instanceof ModelBase) {
                data = BatchOperator.fromTable(((ModelBase) transformers[i]).getModelData())
                    .setMLEnvironmentId(envId);
                data = data.link(new VectorSerializeBatchOp().setMLEnvironmentId(envId));
            }
            if (data != null) {
                // 这对应模型文件中的1，为什么模型文件没有 0就是因为VectorAssembler没有自己的数据，所以就
不包括了。
                packed = new UnionAllBatchOp().setMLEnvironmentId(envId).linkFrom(packed, packB
atchOp(data, i));
            }
        }
        return packed;
    }
}
```

0x05 读取模型

下面代码作用是：读取模型，然后进行转换。

```
BatchOperator data = new CsvSourceBatchOp().setFilePath(URL).setSchemaStr(SCHEMA_STR);
PipelineModel pipeline = PipelineModel.load("./kmeans.csv");
pipeline.transform(data).print();
```

读取模型文件，然后转换成PipelineModel。

```
public class PipelineModel extends ModelBase<PipelineModel> implements LocalPredictable {
    //Load the pipeline model from a path.
```

```
    public static PipelineModel load(String path) {
        return load(new CsvSourceBatchOp(path, PIPELINE_MODEL_SCHEMA));
    }

    //Load the pipeline model from a BatchOperator.
    public static PipelineModel load(BatchOperator batchOp) {
        return new PipelineModel(ModelExporterUtils.unpackTransformersArray(batchOp));
    }

    public PipelineModel(TransformerBase[] transformers) {
        super(null);
        if (null == transformers) {
            this.transformers = new TransformerBase[]{};
        } else {
            List<TransformerBase> flattened = new ArrayList<>();
            flattenTransformers(transformers, flattened);
            this.transformers = flattened.toArray(new TransformerBase[0]);
        }
    }
}
// 相关调用栈如下
unpackTransformersArray:91, ModelExporterUtils (com.alibaba.alink.pipeline)
load:149, PipelineModel (com.alibaba.alink.pipeline)
load:142, PipelineModel (com.alibaba.alink.pipeline)
main:22, KMeansExample2 (com.alibaba.alink)
```

以下是为导入导出用到的功能类，比如导入导出transformer。我们能够看到大致功能如下：

- 从index为-1处获取配置信息。
- 从配置信息中获取了算法类，参数，shema等信息。
- 根据算法类，生成所有transformer。
- 每次生成一个新transformer时候，会读取文件中对应行内容，unpack该行内容，生成模型对应的数据，然后赋值给transformer。注意的是，解析出来的数据被包装成一个BatchOperator。

```
class ModelExporterUtils {
    // Unpack transformers array from a BatchOperator.
    static TransformerBase[] unpackTransformersArray(BatchOperator batchOp) {
        String configStr;
        try {
            // 从index为-1处获取配置信息。
            List<Row> rows = batchOp.as(new String[]{"f1", "f2"}).where("f1=-1").collect();
            Preconditions.checkArgument(rows.size() == 1, "Invalid model.");
            configStr = (String) rows.get(0).getField(1);
        } catch (Exception e) {
            throw new RuntimeException("Fail to collect model config.");
        }
        // 这里从配置信息中获取了算法类，参数，shema等信息
        String[] clazzNames = JsonConverter.fromJson(JsonPath.read(configStr, "$.clazz").toStri
ng(), String[].class);
        String[] params = JsonConverter.fromJson(JsonPath.read(configStr, "$.param").toString()
, String[].class);
        String[] schemas = JsonConverter.fromJson(JsonPath.read(configStr, "$.schema").toString
(), String[].class);

        // 遍历，生成所有transformer。
        int numTransformers = clazzNames.length;
        TransformerBase[] transformers = new TransformerBase[numTransformers];
        for (int i = 0; i < numTransformers; i++) {
            try {
```

```
                Class clazz = Class.forName(clazzNames[i]);
                transformers[i] = (TransformerBase) clazz.getConstructor(Params.class).newInsta
nce(
                    Params.fromJson(params[i])
                        .set(HasMLEnvironmentId.ML_ENVIRONMENT_ID, batchOp.getMLEnvironmentId()
));
            } catch (Exception e) {
                throw new RuntimeException("Fail to re construct transformer.", e);
            }

            BatchOperator packed = batchOp.as(new String[]{"f1", "f2"}).where("f1=" + i);
            if (transformers[i] instanceof PipelineModel) {
                BatchOperator data = unpackBatchOp(packed, CsvUtil.schemaStr2Schema(schemas[i])
);
                transformers[i] = new PipelineModel(unpackTransformersArray(data))
                    .setMLEnvironmentId(batchOp.getMLEnvironmentId());
            } else if (transformers[i] instanceof ModelBase) {
                BatchOperator data = unpackBatchOp(packed, CsvUtil.schemaStr2Schema(schemas[i])
);
                // 这里会设置模型数据。
                ((ModelBase) transformers[i]).setModelData(data.getOutputTable());
            }
        }
        return transformers;
    }

}
```

最后生成的transformers如下：

```
transformers = {TransformerBase[2]@9340}
 0 = {VectorAssembler@9383}
  mapperBuilder = {VectorAssembler$lambda@9385}
  params = {Params@9386} "Params {outputCol="features", selectedCols=["sepal_length","sepal_wid
th","petal_length","petal_width"], MLEnvironmentId=0}"
 1 = {KMeansModel@9384}
  mapperBuilder = {KMeansModel$lambda@9388}
  modelData = {TableImpl@9389} "UnnamedTable$1"
  params = {Params@9390} "Params {vectorCol="features", maxIter=100, reservedCols=["category"],
 k=3, MLEnvironmentId=0, predictionCol="prediction_result", predictionDetailCol="prediction_det
ail"}"
```

0x06 预测

`pipeline.transform(data).print();` 是预测的代码。

6.1 生成runtime rapper

预测算法需要被包装成RichMapFunction，才能够被Flink引用。

VectorAssembler是起到转换csv文件作用。KMeansModel是用来预测。预测时候会调用到KMeansModel.transform，其又会调用到linkFrom，这里生成了runtime rapper。

```
public abstract class MapModel<T extends MapModel<T>>
                extends ModelBase<T> implements LocalPredictable {
                @Override
    public BatchOperator transform(BatchOperator input) {
        return new ModelMapBatchOp(this.mapperBuilder, this.params)
              .linkFrom(BatchOperator.fromTable(this.getModelData())
```

```
                .setMLEnvironmentId(input.getMLEnvironmentId()), input);
    }
}

// this.getModelData()是模型数据，对应linkFrom的输入参数inputs[0]
// input 这个是待处理的数据。，对应linkFrom的输入参数inputs[1]

// 模型数据就是之前从csv中取出来设置的。
public abstract class ModelBase<M extends ModelBase<M>> extends TransformerBase<M>
    implements Model<M> {
    public Table getModelData() {
        return this.modelData;
    }
}
```

ModelMapBatchOp.linkFrom 代码中，会生成ModelMapperAdapter。此时会把模型信息作为广播变量存起来。这样在后续预测时候就可以先load模型数据。

```
public class ModelMapBatchOp<T extends ModelMapBatchOp<T>> extends BatchOperator<T> {

   private static final String BROADCAST_MODEL_TABLE_NAME = "broadcastModelTable";

   // (modelScheme, dataSchema, params) -> ModelMapper
   private final TriFunction<TableSchema, TableSchema, Params, ModelMapper> mapperBuilder;

   public ModelMapBatchOp(TriFunction<TableSchema, TableSchema, Params, ModelMapper> mapperBuil
der, Params params) {
      super(params);
      this.mapperBuilder = mapperBuilder;
   }

   @Override
   public T linkFrom(BatchOperator<?>... inputs) {
         BroadcastVariableModelSource modelSource = new BroadcastVariableModelSource(BROADCAST_
MODEL_TABLE_NAME);
         ModelMapper mapper = this.mapperBuilder.apply(
               inputs[0].getSchema(),
               inputs[1].getSchema(),
               this.getParams());
         DataSet<Row> modelRows = inputs[0].getDataSet().rebalance();
         // 这里会广播变量
         DataSet<Row> resultRows = inputs[1].getDataSet()
               .map(new ModelMapperAdapter(mapper, modelSource))
               .withBroadcastSet(modelRows, BROADCAST_MODEL_TABLE_NAME);

         TableSchema outputSchema = mapper.getOutputSchema();
         this.setOutput(resultRows, outputSchema);
         return (T) this;
   }
}
```

6.2 加载模型

当预测时候，ModelMapperAdapter会在open函数先加载模型。

```
public class ModelMapperAdapter extends RichMapFunction<Row, Row> implements Serializable {
    @Override
    public void open(Configuration parameters) throws Exception {
        List<Row> modelRows = this.modelSource.getModelRows(getRuntimeContext());
```

```
        this.mapper.loadModel(modelRows);
    }
}

// 加载出来的模型数据举例如下
modelRows = {ArrayList@10100}  size = 4
 0 = {Row@10103} "2097152,{"clusterId":1,"weight":62.0,"vec":{"data":[5.901612903225806,2.74838
70967741932,4.393548387096773,1.4338709677419355]}}"
 1 = {Row@10104} "0,{"vectorCol":"\"features\"","latitudeCol":null,"longitudeCol":null,"distanc
eType":"\"EUCLIDEAN\"","k":"3","vectorSize":"4"}"
 2 = {Row@10105} "3145728,{"clusterId":2,"weight":50.0,"vec":{"data":[5.005999999999999,3.418,1
.4639999999999997,0.24400000000000002]}}"
 3 = {Row@10106} "1048576,{"clusterId":0,"weight":38.0,"vec":{"data":[6.85,3.0736842105263156,5
.742105263157894,2.0710526315789477]}}"
```

this.mapper.loadModel(modelRows) 会调用KMeansModelMapper.loadModel，其最后调用到

- ModelConverterUtils.extractModelMetaAndData 来进行反序列化，把DataSet转换回Tuple。
- 最终调用到KMeansUtil.KMeansTrainModelData生成用来预测的模型KMeansTrainModelData

```
/**
 * The abstract class for a kind of {@link ModelDataConverter} where the model data can seriali
ze to "Tuple2&jt;Params, Iterable&jt;String&gt;&gt;". Here "Params" is the meta data of the mod
el, and "Iterable&jt;String&gt;" is concrete data of the model.
 */
public abstract class SimpleModelDataConverter<M1, M2> implements ModelDataConverter<M1, M2> {
    @Override
    public M2 load(List<Row> rows) {
        Tuple2<Params, Iterable<String>> metaAndData = ModelConverterUtils.extractModelMetaAndD
ata(rows);
        return deserializeModel(metaAndData.f0, metaAndData.f1);
    }
}

metaAndData = {Tuple2@10267} "(Params {vectorCol="features", latitudeCol=null, longitudeCol=nul
l, distanceType="EUCLIDEAN", k=3, vectorSize=4},com.alibaba.alink.common.model.ModelConverterUt
ils$StringDataIterable@7e9c1b42)"
 f0 = {Params@10252} "Params {vectorCol="features", latitudeCol=null, longitudeCol=null, distan
ceType="EUCLIDEAN", k=3, vectorSize=4}"
  params = {HashMap@10273}  size = 6
   "vectorCol" -> ""features""
   "latitudeCol" -> null
   "longitudeCol" -> null
   "distanceType" -> ""EUCLIDEAN""
   "k" -> "3"
   "vectorSize" -> "4"
 f1 = {ModelConverterUtils$StringDataIterable@10262}
  iterator = {ModelConverterUtils$StringDataIterator@10272}
   modelRows = {ArrayList@10043}  size = 4
   order = {Integer[4]@10388}
   curr = "{"clusterId":0,"weight":38.0,"vec":{"data":[6.85,3.0736842105263156,5.74210526315789
4,2.0710526315789477]}}"
   listPos = 2
```

可以看到getModelRows就是从广播变量中读取数据。

```
public class BroadcastVariableModelSource implements ModelSource {
    public List<Row> getModelRows(RuntimeContext runtimeContext) {
        return runtimeContext.getBroadcastVariable(modelVariableName);
```

```
    }
}
```

6.3 预测

最后预测是在ModelMapperAdapter的map函数。这实际上是 flink根据用户代码生成的执行计划进行相应处理后自己执行的。

```
/**
 * Adapt a {@link ModelMapper} to run within flink.
 * <p>
 * This adapter class hold the target {@link ModelMapper} and it's {@link ModelSource}. Upon open(),
 * it will load model rows from {@link ModelSource} into {@link ModelMapper}.
 */
public class ModelMapperAdapter extends RichMapFunction<Row, Row> implements Serializable {
    @Override
    public Row map(Row row) throws Exception {
        return this.mapper.map(row);
    }
}
```

mapper实际调用到KMeansModelMapper，这里就用到了模型数据。

```
// Find  the closest cluster center for every point.
public class KMeansModelMapper extends ModelMapper {
    @Override
    public Row map(Row row){
        Vector record = KMeansUtil.getKMeansPredictVector(colIdx, row);
            ......
            if(isPredDetail){
                double[] probs = KMeansUtil.getProbArrayFromDistanceArray(clusterDistances);
                DenseVector vec = new DenseVector(probs.length);
                for(int i = 0; i < this.modelData.params.k; i++){
                    // 这里就用到了模型数据进行预测
                    vec.set((int)this.modelData.getClusterId(i), probs[i]);
                }
                res.add(vec.toString());
            }
      return outputColsHelper.getResultRow(row, Row.of(res.toArray(new Object[0])));
                }
}

// 模型数据如下
this = {KMeansModelMapper@10822}
 modelData = {KMeansPredictModelData@10828}
  centroids = {FastDistanceMatrixData@10842}
   vectors = {DenseMatrix@10843} "mat[4,3]:\n  5.006,6.85,5.901612903225807\n  3.418,3.0736842105263156,2.7483870967741937\n  1.4639999999999997,5.742105263157894,4.393548387096774\n  0.244000000000000002,2.0710526315789473,1.4338709677419355\n"
   label = {DenseMatrix@10844} "mat[1,3]:\n  38.945592000000005,93.63106648199445,63.74191987513008\n"
   rows = {Row[3]@10845}
  params = {KMeansTrainModelData$ParamSummary@10829}
   k = 3
   vectorSize = 4
   distanceType = {DistanceType@10849} "EUCLIDEAN"
   vectorColName = "features"
```

```
    latitudeColName = null
    longtitudeColName = null
```

0x07 流式预测

我们知道Alink是可以支持批式预测和流式预测。我们看看流式预测是怎么处理的。下面就是KMeans的流式预测。

```
public class KMeansExampleStream {
    AlgoOperator getData(boolean isBatch) {
        Row[] array = new Row[] {
                Row.of(0, "0 0 0"),
                Row.of(1, "0.1,0.1,0.1"),
                Row.of(2, "0.2,0.2,0.2"),
                Row.of(3, "9 9 9"),
                Row.of(4, "9.1 9.1 9.1"),
                Row.of(5, "9.2 9.2 9.2")
        };

        if (isBatch) {
            return new MemSourceBatchOp(
                    Arrays.asList(array), new String[] {"id", "vec"});
        } else {
            return new MemSourceStreamOp(
                    Arrays.asList(array), new String[] {"id", "vec"});
        }
    }

    public static void main(String[] args) throws Exception {
        KMeansExampleStream ks = new KMeansExampleStream();
        BatchOperator inOp1 = (BatchOperator)ks.getData(true);
        StreamOperator inOp2 = (StreamOperator)ks.getData(false);

        KMeansTrainBatchOp trainBatch = new KMeansTrainBatchOp().setVectorCol("vec").setK(2);
        KMeansPredictBatchOp predictBatch = new KMeansPredictBatchOp().setPredictionCol("pred")
;

        trainBatch.linkFrom(inOp1);
        KMeansPredictStreamOp predictStream = new KMeansPredictStreamOp(trainBatch).setPredicti
onCol("pred");
        predictStream.linkFrom(inOp2);
        predictStream.print(-1,5);
        StreamOperator.execute();
    }
}
```

`predictStream.linkFrom` 是我们这里的要点，其调用到ModelMapStreamOp。ModelMapStreamOp这个类的代码虽然少，但是条理非常清晰，非常适合学习。

- 首先相关继承关系如下 `KMeansPredictStreamOp extends ModelMapStreamOp`
- 其次能看出来，流预测所依赖的数据模型依然是一个批处理产生的模型 `BatchOperator model` 。
- mapperBuilder是业务模型算子，其构造是通过(modelScheme, dataSchema, params) 得出来的，这恰恰就是机器学习的几个要素。
- KMeansModelMapper就是具体模型算子： `KMeansModelMapper extends ModelMapper` 。

```
// Find  the closest cluster center for every point.
public final class KMeansPredictStreamOp extends ModelMapStreamOp <KMeansPredictStreamOp>
    implements KMeansPredictParams <KMeansPredictStreamOp> {
```

```
   // @param model trained from kMeansBatchOp
   public KMeansPredictStreamOp(BatchOperator model) {
      this(model, new Params());
   }

   public KMeansPredictStreamOp(BatchOperator model, Params params) {
      super(model, KMeansModelMapper::new, params);
   }
}
```

具体深入代码，我们可以看到：

- 首先，把DataSet的数据一次性都取出来，因为都取出来容易造成内存问题，所以 DataSet.collect 注释中有警告：Convenience method to get the elements of a DataSet as a List. As DataSet can contain a lot of data, this method should be used with caution.
- 其次，通过如下代码 `this.mapperBuilder.apply(modelSchema, in.getSchema(), this.getParams());` 构建业务模型KMeansModelMapper。
- 然后， `new ModelMapperAdapter(mapper, modelSource)` 会建立一个 RichFunction 作为运行适配层。
- 最后，输入的流数据源 in 会通过 `in.getDataStream().map((new ModelMapperAdapter(mapper, modelSource));` 来完成预测。
- 实际上，这时候只是生成stream graph，具体计算是后续flink会根据graph再进行处理。

```
public class ModelMapStreamOp<T extends ModelMapStreamOp <T>> extends StreamOperator<T> {

        private final BatchOperator model;
        // (modelScheme, dataSchema, params) -> ModelMapper
        private final TriFunction<TableSchema, TableSchema, Params, ModelMapper> mapperBuilder;

        public ModelMapStreamOp(BatchOperator model,
                                                        TriFunction<TableSchema, TableSchema, P
arams, ModelMapper> mapperBuilder,
                                                        Params params) {
                super(params);
                this.model = model;
                this.mapperBuilder = mapperBuilder;
        }

        @Override
        public T linkFrom(StreamOperator<?>... inputs) {
                StreamOperator<?> in = checkAndGetFirst(inputs);
                TableSchema modelSchema = this.model.getSchema();

                try {
      // 把模型数据全都取出来
                        DataBridge modelDataBridge = DirectReader.collect(model);
                        DataBridgeModelSource modelSource = new DataBridgeModelSource(modelData
Bridge);
                        ModelMapper mapper = this.mapperBuilder.apply(modelSchema, in.getSchema
(), this.getParams());
      // 生成runtime适配层和预测算子。把预测结果返回。
      // 实际上，这时候只是生成stream graph，具体计算是后续flink会根据graph再进行处理。
                        DataStream <Row> resultRows = in.getDataStream().map(new ModelMapperAda
pter(mapper, modelSource));
                        TableSchema resultSchema = mapper.getOutputSchema();
                        this.setOutput(resultRows, resultSchema);

                        return (T) this;
```

```
                } catch (Exception ex) {
                        throw new RuntimeException(ex);
                }
        }
}
```

0x08 总结

现在我们已经梳理了Alink模型的来龙去脉，让我们再次拿出模型文件内容来验证。

- 第一行是元数据信息，其中包含schema，算法类名称，元参数。Alink可以通过这些信息生成流水线的transformer。
- 后续行是算法类所需要的模型数据。每一行对应一个算法类。Alink会取出这些数据来设置到transformer中。
- 后续行的模型数据是具体算法相关。
- 第一行特殊之处在于其index是 -1。后续数据行的index从0开始，如果某一个transformer没有数据，则没有对应行，跳过index。

这样Alink就可以根据模型文件生成流水线模型。

```
-1,"{""schema"":[""""],""model_id BIGINT,model_info VARCHAR""],""param"":[""{\""outputCol\"":\""
\\\""features\\\""\"",\""selectedCols\"":\""[\\\""sepal_length\\\"",\\\""sepal_width\\\"",\\\""
petal_length\\\"",\\\""petal_width\\\""]\""}"",""{\""vectorCol\"":\""\\\""features\\\""\"",\""m
axIter\"":\""100\"",\""reservedCols\"":\""[\\\""category\\\""]\"",\""k\"":\""3\"",\""prediction
Col\"":\""\\\""prediction_result\\\""\"",\""predictionDetailCol\"":\""\\\""prediction_detail\\\
""\""}""],""clazz"":[""com.alibaba.alink.pipeline.dataproc.vector.VectorAssembler"",""com.aliba
ba.alink.pipeline.clustering.KMeansModel""]}"

1,"0^{""vectorCol"":""\""features\"""",""latitudeCol"":null,""longitudeCol"":null,""distanceTyp
e"":""\""EUCLIDEAN\"""",""k"":""3"",""vectorSize"":""4""}"
1,"1048576^{""clusterId"":0,""weight"":39.0,""vec"":{""data"":[6.8538461538461535,3.07692307692
30766,5.7153846153846155,2.0538461538461545]}}"
1,"2097152^{""clusterId"":1,""weight"":61.0,""vec"":{""data"":[5.883606557377049,2.740983606557
377,4.388524590163936,1.4344262295081969]}}"
1,"3145728^{""clusterId"":2,""weight"":50.0,""vec"":{""data"":[5.006,3.418,1.4640000000000002,0
.24400000000000005]}}"
```