Alink漫谈(一)：从KMeans算法实现不同看Alink设计思想

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。本文将带领大家从多重角度出发来分析推测Alink的设计思路。

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 Flink 是什么

Apache Flink是由Apache软件基金会开发的开源流处理框架，它通过实现了 Google Dataflow 流式计算模型实现了高吞吐、低延迟、高性能兼具实时流式计算框架。

其核心是用Java和Scala编写的分布式流数据流引擎。Flink以数据并行和流水线方式执行任意流数据程序，Flink的流水线运行时系统可以执行批处理和流处理程序。此外，Flink的运行时本身也支持迭代算法的执行。

0x02 Alink 是什么

Alink 是阿里巴巴计算平台事业部PAI团队从2017年开始基于实时计算引擎 Flink 研发的新一代机器学习算法平台，提供丰富的算法组件库和便捷的操作框架，开发者可以一键搭建覆盖数据处理、特征工程、模型训练、模型预测的算法模型开发全流程。项目之所以定为Alink，是取自相关名称（Alibaba, Algorithm, AI, Flink, Blink）的公共部分。

借助Flink在批流一体化方面的优势，Alink能够为批流任务提供一致性的操作。在2017年初，阿里团队通过调研团队看到了Flink在批流一体化方面的优势及底层引擎的优秀性能，于是基于Flink重新设计研发了机器学习算法库，即Alink平台。该平台于2018年在阿里集团内部上线，随后不断改进完善，在阿里内部错综复杂的业务场景中锻炼成长。

0x03 Alink设计思路

因为目前关于Alink设计的公开资料比较少，我们手头只有其源码，看起来只能从代码反推。但是世界上的事物都不是孤立的，我们还有其他角度来帮助我们判断推理。所以下面就让我们来进行推断。

1. 白手起家

FlinkML 是 Flink 社区现存的一套机器学习算法库，这一套算法库已经存在很久而且更新比较缓慢。

Alink团队起初面临的抉择是：是否要基于 Flink ML 进行开发，或者对 Flink ML进行更新。

经过研究，Alink团队发现，Flink ML 其仅支持10余种算法，支持的数据结构也不够通用，在算法性能方面做的优化也比较少，而且其代码也很久没有更新。所以，他们放弃了基于旧版FlinkML进行改进、升级的想法，决定基于Flink重新设计研发机器学习算法库。

所以我们要分析的就是如何从无到有设计出一个新的机器学习平台/框架。

2. 替代品如何造成威胁

因为Alink是市场的新进入者，所以Alink的最大问题就是如何替代市场上的现有产品。

迈克尔·波特用“*替代品威胁*”来解释用户的整个替代逻辑，当新产品能牢牢掌握住这一点，就有可能在市场上获得非常好的表现，打败竞争对手。

假如现在想从0到1构建一个机器学习库或者机器学习框架，那么我们需要从商业意识和商业逻辑出发，来思考这个产品的价值所在，就能对这个产品做个比较精确的定义，从而能够确定产品路线。

产品需要解决应用环境下的综合性问题，产品的价值体现，可以分拆了三个维度。

- **用户的角度**：价值体现在用户使用，获取产品的意愿。这个就是换用成本的问题，一旦换用成本过高，这个产品就很难成功。
- **竞争对手的角度**：产品的竞争力，最终都体现为用户为了获取该产品愿意支付的最高成本上限，当一个替代品进入市场，必须有能给用户足够的洞理驱使用户换用替代品。
- **企业的角度**：站在企业的角度，实际就是成本结构和收益的规模性问题 。

下面就让我们逐一分析。

3. 用户角度看设计

这个就是换用成本的问题，一旦换用成本过高，这个产品就很难成功。

Alink大略有两种用户：算法工程师，应用工程师。

Alink算法工程师特指实现机器学习算法的工程师。Alink应用工程师就是应用Alink AI算法做业务的工程师。这两类用户的换用成本都是Alink需要考虑的。

新产品对于用户来说，有两个大的问题：产品底层逻辑和开发工具。一个优秀的新产品绝对不能在这两个问题上增加用户的换用成本。

底层逻辑Flink

Flink这个平台博大精深，无论是熟悉其API还是深入理解系统架构都不是容易的事情。如果Alink用户还需要熟悉Flink，那势必造成ALink用户的换用成本，所以这点应该尽量避免。

- 对于算法工程师，他们应该主要把思路集中在算法上，而尽量不用关心Flink内部的细节，如果一定要熟悉Flink，那么越少越好；
- 对于应用工程师，他们主要的需求就是API接口越简单越好，他们最理想的状态应该是：完全感觉不到Flink的存在。

综上所述，**Alink的原则之一应该是：算法的归算法，Flink的归Flink，尽量屏蔽AI算法和Flink之间的联系**。

开发工具

开发工具就是究竟用什么语言开发。Flink的开发语言主要是JAVA，SCALA，Python。而机器学习世界中主要还是Python。

- 首先要排除SCALA。因为Scala 是一门很难掌握的语言，它的规则是基于数学类型理论的，学习曲线相当陡峭。一个能够领会规则和语言特性的优秀程序员，使用 Scala 会比使用 Java 更高效，但是一个普通程序员的生产力，从功能实现上来看，效率则会相反。

  让我们看看基于Flink的原生KMeans SCALA代码，很多人看了之后恐怕都会懵圈。

```
    val finalCentroids = centroids.iterate(params.getInt("iterations", 10)) { currentCentroids => val newCentroids = points
        .map(new SelectNearestCenter).withBroadcastSet(currentCentroids, "centroids")
        .map { x => (x._1, x._2, 1L) }.withForwardedFields("_1; _2")
        .groupBy(0)
        .reduce { (p1, p2) => (p1._1, p1._2.add(p2._2), p1._3 + p2._3) }.withForwardedFields("_1")
        .map { x => new Centroid(x._1, x._2.div(x._3)) }.withForwardedFields("_1->id")
      newCentroids
    }
```

- 其次是选择JAVA还是Python开发具体算法。Alink内部肯定进行了很多权宜和抉择。因为这个不单单是哪个语言本身更合适，也涉及到Alink团队内部有哪些资源，比如是JAVA工程师更多还是Python更多。最终Alink选择了JAVA来开发算法。
- 最后是API。这个就没有什么疑问了，Alink提供了Python和JAVA两种语言的API，直接可参见GitHub的介绍。

> 在 PyAlink 中，算法组件提供的接口基本与 Java API 一致，即通过默认构造方法创建一个算法组件，然后通过 `setXXX` 设置参数，通过 `link/linkTo/linkFrom` 与其他组件相连。 这里利用 Jupyter 的自动补全机制可以提供书写便利。

另外，如果采用JAVA或者Python，肯定有大量现有代码可以修改复用。如果采用SCALA，就难以复用之前的积累。

综上所述，**Alink的原则之一应该是：采用最简单，最常见的开发语言和设计思维**。

4. 竞争对手角度看设计

Alink的竞争对手大略可以认为是Spark ML, Flink ML, *Scikit*-learn。

他们是市场上的现有力量，拥有大量的用户。用户已经熟悉了这些竞争对手的设计思路，开发策略，基本概念和API。除非Alink能够提供一种神奇简便的API，否则Alink应该在设计上最大程度借鉴这些竞争对手。

比如机器学习开发中有如下常见概念：Transformer，Estimator，PipeLine，Parameter。这些概念 Alink 应该尽量提供。

综上所述，**Alink的原则之一应该是 ： 尽量借鉴市面上通用的设计思路和开发模式，让开发者无缝切换 **。

从 Alink的目录结构中 ，我们可以看出，Alink确实提供了这些常见概念。

比如 Pipeline，Trainer，Model，Estimator。我们会在后续文章中再详细介绍这些概念。

```
./java/com/alibaba/alink:
common          operator        params          pipeline

./java/com/alibaba/alink/params:
associationrule evaluation      nlp             regression      statistics
classification  feature         onlinelearning  shared          tuning
clustering      io              outlier         similarity      udf
dataproc        mapper          recommendation  sql             validators

./java/com/alibaba/alink/pipeline:
EstimatorBase.java      ModelBase.java          Trainer.java            feature
LocalPredictable.java   ModelExporterUtils.java TransformerBase.java    nlp
LocalPredictor.java     Pipeline.java           classification          recommendation
MapModel.java           PipelineModel.java      clustering              regression
MapTransformer.java     PipelineStageBase.java  dataproc                tuning
```

5. 企业角度看设计

这是成本结构和收益的规模性问题。从而决定了Alink在开发时候，必须尽量提高开发工程师的效率，提高生产力。前面提到的弃用SCALA，部分也出于这个考虑。

挑战集中在：

- 如何在对开发者最大程度屏蔽Flink的情况下，依然利用好Flink的各种能力。
- 如何构建一套相应打法和战术体系，即middleware或者adapter，让用户基于此可以快速开发算法

举个例子：

- 肯定有个别开发者，其对Flink特别熟悉，他们可以运用各种Flink API和函数编程思维开发出高效率的算法。这种开发者，我们可以称为是武松武都头。他们类似特种兵，能上战场冲锋陷阵，也能吊打白额大虫。
- 但是绝大多数开发者对Flink不熟悉，他们更熟悉AI算法和命令式编程思路。这种开发者我们可以认为他们属于八十万禁军或者是玄甲军，北府兵，魏武卒，背嵬军。这种才是实际开发中的主力部队和常规套路。

我们需要针对八十万禁军，让林冲林教头设计出一套适合正规作战的枪棒打法。或者针对背嵬军，让岳飞岳元帅设计一套马军冲阵机制。

因此，**Alink的原则之一应该是 ： 构建一套战术打法（middleware或者adapter），即屏蔽了Flink，又可以利用好Flink，还可以让用户基于此可以快速开发算法 **。

我们想想看大概有哪些基础工作需要做：

- 如何初始化
- 如果通信
- 如何分割代码，如何广播代码
- 如果分割数据，如何广播数据
- 如何迭代算法

- ……

让我们看看Alink做了哪些努力，这点从其目录结构可以看出有queue，operator，mapper等等构建架构所必须的数据结构：

```
./java/com/alibaba/alink/common:
MLEnvironment.java              linalg MLEnvironmentFactory.java        mapper
VectorTypes.java                model comqueue                  utils io

./java/com/alibaba/alink/operator:
AlgoOperator.java       common  batch    stream
```

其中最重要的概念是BaseComQueue，这是把通信或者计算抽象成ComQueueItem，然后把ComQueueItem串联起来形成队列。这样就形成了面向迭代计算场景的一套迭代通信计算框架。其他数据结构大多是围绕着BaseComQueue来具体运作。

```
/**
 * Base class for the com(Computation && Communicate) queue.
 */
public class BaseComQueue<Q extends BaseComQueue<Q>> implements Serializable {
        /**
         * All computation or communication functions.
         */
        private final List<ComQueueItem> queue = new ArrayList<>();
        /**
         * sessionId for shared objects within this BaseComQueue.
         */
        private final int sessionId = SessionSharedObjs.getNewSessionId();
        /**
         * The function executed to decide whether to break the loop.
         */
        private CompareCriterionFunction compareCriterion;
        /**
         * The function executed when closing the iteration
         */
        private CompleteResultFunction completeResult;
        /**
         * Max iteration count.
         */
        private int maxIter = Integer.MAX_VALUE;

        private transient ExecutionEnvironment executionEnvironment;
}
```

MLEnvironment 是另外一个重要的类。其封装了Flink开发所必须要的运行上下文。用户可以通过这个类来获取各种实际运行环境，可以建立table，可以运行SQL语句。

```
/**
 * The MLEnvironment stores the necessary context in Flink.
 * Each MLEnvironment will be associated with a unique ID.
 * The operations associated with the same MLEnvironment ID
 * will share the same Flink job context.
 */
public class MLEnvironment {
    private ExecutionEnvironment env;
    private StreamExecutionEnvironment streamEnv;
    private BatchTableEnvironment batchTableEnv;
    private StreamTableEnvironment streamTableEnv;
}
```

6. 设计原则总结

下面我们可以总结下Alink部分设计原则

- 算法的归算法，Flink的归Flink，尽量屏蔽AI算法和Flink之间的联系。
- 采用最简单，最常见的开发语言。
- 尽量借鉴市面上通用的设计思路和开发模式，让开发者无缝切换。
- 构建一套战术打法（middleware或者adapter），即屏蔽了Flink，又可以利用好Flink，还可以让用户基于此可以快速开发算法。

0x04 KMeans算法实现看设计

Flink和Alink源码中，都提供了KMeans算法例子，所以我们就从KMeans入手看看Flink原生算法和Alink算法实现的区别。为了统一标准，我们都选用JAVA版本的算法实现。

1. KMeans算法

KMeans算法的思想比较简单，假设我们要把数据分成K个类，大概可以分为以下几个步骤：

- 随机选取k个点，作为聚类中心；
- 计算每个点分别到k个聚类中心的聚类，然后将该点分到最近的聚类中心，这样就行成了k个簇；
- 再重新计算每个簇的质心（均值）；
- 重复以上2~4步，直到质心的位置不再发生变化或者达到设定的迭代次数。

2. Flink KMeans例子

> K-Means 是迭代的聚类算法，初始设置K个聚类中心
>
> 1. 在每一次迭代过程中，算法计算每个数据点到每个聚类中心的欧式距离
> 2. 每个点被分配到它最近的聚类中心
> 3. 随后每个聚类中心被移动到所有被分配的点
> 4. 移动的聚类中心被分配到下一次迭代
> 5. 算法在固定次数的迭代之后终止(在本实现中，参数设置)
> 6. 或者聚类中心在迭代中不在移动
> 7. 本项目是工作在二维平面的数据点上
> 8. 它计算分配给集群中心的数据点
> 9. 每个数据点都使用其所属的最终集群(中心)的id进行注释。

下面给出部分代码，具体算法解释可以在注释中看到。

这里主要采用了Flink的批量迭代。其调用 DataSet 的 `iterate(int)` 方法创建一个 BulkIteration，迭代以此为起点，返回一个 IterativeDataSet，可以用常规运算符进行转换。迭代调用的参数 int 指定最大迭代次数。

IterativeDataSet 调用 `closeWith(DataSet)` 方法来指定哪个转换应该反馈到下一个迭代，可以选择使用 `closeWith(DataSet, DataSet)` 指定终止条件。如果该 DataSet 为空，则它将评估第二个 DataSet 并终止迭代。如果没有指定终止条件，则迭代在给定的最大次数迭代后终止。

```
public class KMeans {

        public static void main(String[] args) throws Exception {

                // Checking input parameters
                final ParameterTool params = ParameterTool.fromArgs(args);
```

```
                // set up execution environment
                ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
                env.getConfig().setGlobalJobParameters(params); // make parameters available in
 the web interface

                // get input data:
                // read the points and centroids from the provided paths or fall back to defaul
t data
                DataSet<Point> points = getPointDataSet(params, env);
                DataSet<Centroid> centroids = getCentroidDataSet(params, env);

                // set number of bulk iterations for KMeans algorithm
                IterativeDataSet<Centroid> loop = centroids.iterate(params.getInt("iterations",
 10));

                DataSet<Centroid> newCentroids = points
                        // compute closest centroid for each point
                        .map(new SelectNearestCenter()).withBroadcastSet(loop, "centroids")
                        // count and sum point coordinates for each centroid
                        .map(new CountAppender())
                        .groupBy(0).reduce(new CentroidAccumulator())
                        // compute new centroids from point counts and coordinate sums
                        .map(new CentroidAverager());

                // feed new centroids back into next iteration
                DataSet<Centroid> finalCentroids = loop.closeWith(newCentroids);

                DataSet<Tuple2<Integer, Point>> clusteredPoints = points
                        // assign points to final clusters
                        .map(new SelectNearestCenter()).withBroadcastSet(finalCentroids, "centr
oids");

                // emit result
                if (params.has("output")) {
                        clusteredPoints.writeAsCsv(params.get("output"), "\n", " ");
                        // since file sinks are lazy, we trigger the execution explicitly
                        env.execute("KMeans Example");
                } else {
                        System.out.println("Printing result to stdout. Use --output to specify
output path.");
                        clusteredPoints.print();
                }
        }
```

3. Alink KMeans示例

Alink中，Kmeans是分布在若干文件中，这里我们提取部分代码来对照。

KMeansTrainBatchOp

这里是算法主程序，这里倒是看起来十分清爽干净，但实际上是没有这么简单，Alink在其背后做了大量的基础工作。

可以看出，算法实现的主要工作是：

- 构建了一个IterativeComQueue（BaseComQueue的缺省实现）。
- 初始化数据，这里有两种办法：initWithPartitionedData将DataSet分片缓存至内存。initWithBroadcastData将DataSet整体缓存至每个worker的内存。

- 将计算分割为若干ComputeFunction，比如KMeansPreallocateCentroid / KMeansAssignCluster / KMeansUpdateCentroids ...，串联在IterativeComQueue。
- 运用AllReduce通信模型完成了数据同步。

```
public final class KMeansTrainBatchOp extends BatchOperator <KMeansTrainBatchOp>
        implements KMeansTrainParams <KMeansTrainBatchOp> {

        static DataSet <Row> iterateICQ(...省略...) {

                return new IterativeComQueue()
                        .initWithPartitionedData(TRAIN_DATA, data)
                        .initWithBroadcastData(INIT_CENTROID, initCentroid)
                        .initWithBroadcastData(KMEANS_STATISTICS, statistics)
                        .add(new KMeansPreallocateCentroid())
                        .add(new KMeansAssignCluster(distance))
                        .add(new AllReduce(CENTROID_ALL_REDUCE))
                        .add(new KMeansUpdateCentroids(distance))
                        .setCompareCriterionOfNode0(new KMeansIterTermination(distance, tol))
                        .closeWith(new KMeansOutputModel(distanceType, vectorColName, latitudeC
olName, longitudeColName))
                        .setMaxIter(maxIter)
                        .exec();
        }
}
```

KMeansPreallocateCentroid

预先分配聚类中心

```
public class KMeansPreallocateCentroid extends ComputeFunction {
    public void calc(ComContext context) {
        if (context.getStepNo() == 1) {
            List<FastDistanceMatrixData> initCentroids = (List)context.getObj("initCentroid");
            List<Integer> list = (List)context.getObj("statistics");
            Integer vectorSize = (Integer)list.get(0);
            context.putObj("vectorSize", vectorSize);
            FastDistanceMatrixData centroid = (FastDistanceMatrixData)initCentroids.get(0);
            Preconditions.checkArgument(centroid.getVectors().numRows() == vectorSize, "Init ce
ntroid error, size not equal!");
            context.putObj("centroid1", Tuple2.of(context.getStepNo() - 1, centroid));
            context.putObj("centroid2", Tuple2.of(context.getStepNo() - 1, new FastDistanceMatr
ixData(centroid)));
            context.putObj("k", centroid.getVectors().numCols());
        }
    }
}
```

KMeansAssignCluster

为每个点(point)计算最近的聚类中心，为每个聚类中心的点坐标的计数和求和

```
/**
 * Find the closest cluster for every point and calculate the sums of the points belonging to t
he same cluster.
 */
public class KMeansAssignCluster extends ComputeFunction {
    @Override
    public void calc(ComContext context) {

        Integer vectorSize = context.getObj(KMeansTrainBatchOp.VECTOR_SIZE);
```

```
        Integer k = context.getObj(KMeansTrainBatchOp.K);
        // get iterative coefficient from static memory.
        Tuple2<Integer, FastDistanceMatrixData> stepNumCentroids;
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        }

        if (null == distanceMatrix) {
            distanceMatrix = new DenseMatrix(k, 1);
        }

        double[] sumMatrixData = context.getObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE);
        if (sumMatrixData == null) {
            sumMatrixData = new double[k * (vectorSize + 1)];
            context.putObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE, sumMatrixData);
        }

        Iterable<FastDistanceVectorData> trainData = context.getObj(KMeansTrainBatchOp.TRAIN_DA
TA);
        if (trainData == null) {
            return;
        }

        Arrays.fill(sumMatrixData, 0.0);
        for (FastDistanceVectorData sample : trainData) {
            KMeansUtil.updateSumMatrix(sample, 1, stepNumCentroids.f1, vectorSize, sumMatrixDat
a, k, fastDistance,
                distanceMatrix);
        }
    }
}
```

KMeansUpdateCentroids

基于点计数和坐标，计算新的聚类中心。

```
/**
 * Update the centroids based on the sum of points and point number belonging to the same clust
er.
 */
public class KMeansUpdateCentroids extends ComputeFunction {
    @Override
    public void calc(ComContext context) {

        Integer vectorSize = context.getObj(KMeansTrainBatchOp.VECTOR_SIZE);
        Integer k = context.getObj(KMeansTrainBatchOp.K);
        double[] sumMatrixData = context.getObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE);

        Tuple2<Integer, FastDistanceMatrixData> stepNumCentroids;
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        }

        stepNumCentroids.f0 = context.getStepNo();
        context.putObj(KMeansTrainBatchOp.K,
            updateCentroids(stepNumCentroids.f1, k, vectorSize, sumMatrixData, distance));
```

```
    }
}
```

4. 区别

代码量

通过下面的分析可以看出，从实际业务代码量角度说，其实差别不大。

- Flink的代码量少；
- Alink的代码量虽然大，但其本质就是把Flink版本的一些用户定义类分离到自己不同类中，并且有很多读取环境变量的代码；

所以Alink代码只能说比Flink原生实现略大。

耦合度

这里指的是与Flink的耦合度。能看出来Flink的KMeans算法需要大量的Flink类。而Alink被最大限度屏蔽了。

- Flink 算法需要引入的flink类如下

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.api.common.functions.RichMapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.functions.FunctionAnnotation.ForwardedFields;
import org.apache.flink.api.java.operators.IterativeDataSet;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.configuration.Configuration;
```

- Alink 算法需要引入的flink类如下，可以看出来ALink使用的都是基本设施，不涉及算子和复杂API，这样就减少了用户的负担。

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.ml.api.misc.param.Params;
import org.apache.flink.types.Row;
import org.apache.flink.util.Preconditions;
```

编程模式

这是一个主要的区别。

- Flink 使用的是函数式编程。这种范式相对新颖，很多工程师不熟悉。
- Alink 依然使用了命令式编程。这样的好处在于，大量现有算法代码可以复用，也更符合绝大多数工程师的习惯。
- Flink 通过Flink的各种算子完成了操作，比如IterativeDataSet实现了迭代。但这种实现对于不熟悉Flink的工程师是个折磨。
- Alink 基于自己的框架，把计算代码总结成了若干ComputeFunction，然后通过IterativeComQueue完成了具体算法的迭代。这样用户其实对Flink是不需要过多深入理解。

在下一期文章中，将从源码角度分析验证本文的设计思路。

0x05 参考

商业模式的定义——做产品到底是做什么

k-means聚类算法原理简析

flink kmeans聚类算法实现

Spark ML简介之Pipeline，DataFrame，Estimator，Transformer

开源 | 全球首个批流一体机器学习平台

斩获GitHub 2000+ Star，阿里云开源的 Alink 机器学习平台如何跑赢双11数据"博弈"？ | AI 技术生态论

Flink DataSet API