Alink漫谈(十)： 线性回归实现 之 数据预处理

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。本文和下文将介绍线性回归在Alink中是如何实现的，希望可以作为大家看线性回归代码的Roadmap。

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。

本系列目前已有十篇，欢迎大家指点

0x01 概念

1.1 线性回归

线性回归是利用数理统计中回归分析，来确定两种或两种以上变量间相互依赖的定量关系的一种统计分析方法，运用十分广泛。其表达形式为y = w'x+e，e为误差服从均值为0的正态分布。

在线性回归中，目标值与特征之间存在着线性相关的关系。即假设这个方程是一个线性方程，一个多元一次方程。

基本形式：给定由 d 个属性描述的示例 ，线性模型试图学得一个通过属性的线性组合来进行预测的函数，即：

$$f(x) = w_1x_1 + w_2x_2...+w_dx_d + b$$
$$f(x) = w_1 x_1 + w_2 x_2 \ldots + w_d x_d + b$$

其中w为参数，也称为权重，可以理解为x1，x2...和 xd 对f(x)的影响度。

一般形式为：

$$f(x) = w^Tx + b$$
$$f(x) = w^T x + b$$

假如我们依据这个公式来预测 f(x)，公式中的x是我们已知的，然而w，b的取值却不知道，只要我们把w，b的取值求解出来，模型就得以确定。我们就可以依据这个公式来做预测了。

那么如何依据训练数据求解 w 和 b 的最优取值呢？关键是衡量 f 和 y 之间的差别。这就牵扯到另外一个概念：**损失函数（Loss Function）**。

1.2 优化模型

假如有一个模型 f(x)，如何判断这个模型是否优秀？这种定性的判断可以通过一个成为经验误差风险的数值来进行衡量，也就是模型 f 在所有训练样本上所犯错误的总和 E(x)。

我们通过在训练集上最小化经验损失来训练模型。换言之，通过调节 f 的参数 w，使得经验误差风险 E(x) 不断下降，最终达到最小值的时候，我们就获得了一个 “最优” 的模型。

但是如果按照上面的定义，E(x) 是一组示性函数的和，因此是不连续不可导的函数，不易优化。为了解决这个问题，人们提出了**“损失函数”**的概念。损失函数就是和误差函数有一定关系（比如是误差函数的上界），但是具有更好的数学性质（比如连续，可导，凸性等），比较容易进行优化。所以我们就可以对损失函数来优化。

损失函数如果连续可导，所以我们可以用梯度下降法等一阶算法，也可以用牛顿法，拟牛顿法等二阶算法。当优化算法收敛后，我们就得到一个不错的模型。如果损失函数是一个凸函数，我们就可以得到最优模型。

典型的优化方法：

	一阶算法	二阶算法
确定性算法	梯度下降法 投影次梯度下降 近端梯度下降 Frank-Wolfe算法 Nesterov加速算法 坐标下降法 对偶坐标上升法	牛顿法，拟牛顿法
随机算法	随机梯度下降法 随机坐标下降法 随机对偶坐标上升法 随机方差减小梯度法	随机拟牛顿法

所以我们可以知道，优化LinearRegression模型 f 的手段一定是：确定损失函数，用 x，y 作为输入训练以求得损失函数最小值，从而确定 f 的参数 w。过程大致如下：

1. 处理输入，把 x, y 转换成算法需要的格式。
2. 找一个合适的预测函数，一般表示为 **h** 函数，该函数就是我们需要找的分类函数，它用来预测输入数据的判断结果。
3. 构造一个Cost函数（损失函数），该函数表示预测的输出（**h**）与训练数据类别（**y**）之间的偏差，可以是二者之间的差（**h-y**）或者是其他的形式。综合考虑所有训练数据的 “损失”，将Cost求和或者求平均，记为**J(θ)**函数，表示所有训练数据预测值与实际类别的偏差。
4. 显然，损失函数 **J(θ)** 函数的值越小表示预测函数越准确（即**h**函数越准确），所以这一步需要做的是找到 **J(θ)** 函数的最小值。注意，损失函数是关于 **θ** 的函数！也就是说，对于损失函数来讲，**θ**不再是函数的参数，而是损失函数的自变量！
5. 准备模型元数据，建立模型。

1.3 损失函数&目标函数

先概括说明：

- 损失函数：计算的是一个样本的误差；
- 代价函数：是整个训练集上所有样本误差的平均，经常和损失函数混用；
- 目标函数：代价函数 + 正则化项；

再详细阐释：

假设我们用 f(X) 来拟合真实值Y。这个输出的f(X)与真实值Y可能是相同的，也可能是不同的，为了表示我们拟合的好坏，我们就用一个函数来度量拟合的程度。这个函数就称为损失函数(loss function)，或者叫代价函数(cost function)。

损失函数用来衡量算法的运行情况，估量模型的预测值与真实值的不一致程度，是一个非负实值函数，通常使用 L(Y,f(x)) 来表示。损失函数越小，模型的鲁棒性就越好。损失函数是**经验风险函数**的核心部分。

目标函数是一个相关但更广的概念，对于目标函数来说在有约束条件下的最小化就是损失函数（loss function）。

因为f(x)可能会过度学习历史数据，导致它在真正预测时效果会很不好，这种情况称为过拟合(over-fitting)。这样得到的函数会过于复杂。所以我们不仅要让经验风险最小化，还要让**结构风险最小化**。这个时候就定义了一个函数 J(x)，这个函数专门用来度量**模型的复杂度**，在机器学习中也叫正则化(regularization)。常用的有 L1, L2范数。

L1 正则的本质是为模型增加了**“模型参数服从零均值拉普拉斯分布”**这一先验知识。

L2 正则的本质是为模型增加了**“模型参数服从零均值正态分布”**这一先验知识。

L1 正则化增加了所有权重 w 参数的绝对值之和逼迫更多 w 为零，也就是变稀疏（ L2 因为其导数也趋 0, 奔向零的速度不如 L1 给力了）。L1 正则化的引入就是为了完成特征自动选择的光荣使命，它会学习地去掉无用的特征，也就是把这些特征对应的权重置为 0。

L2 正则化中增加所有权重 w 参数的平方之和，逼迫所有 w 尽可能趋向零但不为零（L2 的导数趋于零）。因为在未加入 L2 正则化发生过拟合时，拟合函数需要顾忌每一个点，最终形成的拟合函数波动很大，在某些很小的区间里，函数值的变化很剧烈，也就是某些 w 值非常大。为此，L2 正则化的加入就惩罚了权重变大的趋势。

到这一步我们就可以说我们最终的优化函数是：min(L(Y, f(x) + J(x))，即最优化经验风险和结构风险，而这个函数就被称为**目标函数**。

在回归问题中，通过目标函数来求解最优解，常用的是平方误差（最小二乘线性回归）代价函数。损失函数则是平方损失函数。

1.4 最小二乘法

均方误差是回归任务中最常用的性能度量，因此可以使均方误差最小。基于均方误差最小化来进行模型求解的方法称为“最小二乘法”。在线性回归中，最小二乘法就是找到一条直线，使所有样本到直线的 "欧式距离和" 最小。于是线性回归中损失函数就是平方损失函数。

有了这些基础概念，下面我们就开始动手分析Alink的代码。

0x02 示例代码

首先，我们给出线性回归的示例。

```
public class LinearRegressionExample {
    static Row[] vecrows = new Row[] {
            Row.of("$3$0:1.0 1:7.0 2:9.0", "1.0 7.0 9.0", 1.0, 7.0, 9.0, 16.8),
            Row.of("$3$0:1.0 1:3.0 2:3.0", "1.0 3.0 3.0", 1.0, 3.0, 3.0, 6.7),
            Row.of("$3$0:1.0 1:2.0 2:4.0", "1.0 2.0 4.0", 1.0, 2.0, 4.0, 6.9),
            Row.of("$3$0:1.0 1:3.0 2:4.0", "1.0 3.0 4.0", 1.0, 3.0, 4.0, 8.0)
    };
    static String[] veccolNames = new String[] {"svec", "vec", "f0", "f1", "f2", "label"};
    static BatchOperator vecdata = new MemSourceBatchOp(Arrays.asList(vecrows), veccolNames);
    static StreamOperator svecdata = new MemSourceStreamOp(Arrays.asList(vecrows), veccolNames)
;

    public static void main(String[] args) throws Exception {
        String[] xVars = new String[] {"f0", "f1", "f2"};
        String yVar = "label";
        String vec = "vec";
        String svec = "svec";
        LinearRegression linear = new LinearRegression()
                .setLabelCol(yVar)  // 这里把变量都设置好了，后续会用到
                .setFeatureCols(xVars)
                .setPredictionCol("linpred");

        Pipeline pl = new Pipeline().add(linear);
        PipelineModel model = pl.fit(vecdata);

        BatchOperator result = model.transform(vecdata).select(
                new String[] {"label", "linpred"});

        List<Row> data = result.collect();
    }
}
```

输出是

```
svec|vec|f0|f1|f2|label|linpred
----|---|--|--|--|-----|-------
$3$0:1.0 1:7.0 2:9.0|1.0 7.0 9.0|1.0000|7.0000|9.0000|16.8000|16.8148
$3$0:1.0 1:3.0 2:4.0|1.0 3.0 4.0|1.0000|3.0000|4.0000|8.0000|7.8521
$3$0:1.0 1:3.0 2:3.0|1.0 3.0 3.0|1.0000|3.0000|3.0000|6.7000|6.7739
$3$0:1.0 1:2.0 2:4.0|1.0 2.0 4.0|1.0000|2.0000|4.0000|6.9000|6.959
```

根据前文我们可以知道，在回归问题中，通过优化目标函数来求解最优解，常用的是平方误差（最小二乘线性回归）代价函数。损失函数则是平方损失函数。

对应到Alink，优化函数或者优化器是拟牛顿法的L-BFGS算法，目标函数是UnaryLossObjFunc，损失函数是SquareLossFunc。线性回归训练总体逻辑是LinearRegTrainBatchOp。所以我们下面一一论述。

0x03 整体概述

LinearRegression 训练 用到LinearRegTrainBatchOp，而LinearRegTrainBatchOp的基类是BaseLinearModelTrainBatchOp。所以我们来看BaseLinearModelTrainBatchOp。

```
public class LinearRegression extends Trainer <LinearRegression, LinearRegressionModel> impleme
nts LinearRegTrainParams <LinearRegression>, LinearRegPredictParams <LinearRegression> {
   @Override
   protected BatchOperator train(BatchOperator in) {
      return new LinearRegTrainBatchOp(this.getParams()).linkFrom(in);
   }
}
```

BaseLinearModelTrainBatchOp.linkFrom 代码如下，注释中给出了清晰的逻辑：

大体是：

- 获取算法参数，label信息；
- 准备，转换数据到 Tuple3 format <weight, label, feature vector>；
- 获得统计信息，比如向量大小，均值和方差；
- 对训练数据做标准化和插值；
- 使用L-BFGS算法，通过对损失函数求最小值从而对模型优化；
- 准备模型元数据；
- 建立模型；

```
public T linkFrom(BatchOperator<?>... inputs) {
    BatchOperator<?> in = checkAndGetFirst(inputs);
    // Get parameters of this algorithm.
    Params params = getParams();
    // Get type of processing: regression or not
    boolean isRegProc = getIsRegProc(params, linearModelType, modelName);
    // Get label info : including label values and label type.
    Tuple2<DataSet<Object>, TypeInformation> labelInfo = getLabelInfo(in, params, isRegProc);
    // Transform data to Tuple3 format.//weight, label, feature vector.
    DataSet<Tuple3<Double, Double, Vector>> initData = transform(in, params, labelInfo.f0, isRe
gProc);
    // Get statistics variables : including vector size, mean and variance of train data.
    Tuple2<DataSet<Integer>, DataSet<DenseVector[]>>
        statInfo = getStatInfo(initData, params.get(LinearTrainParams.STANDARDIZATION));
    // Do standardization and interception to train data.
    DataSet<Tuple3<Double, Double, Vector>> trainData = preProcess(initData, params, statInfo.f
1);
    // Solve the optimization problem.
    DataSet<Tuple2<DenseVector, double[]>> coefVectorSet = optimize(params, statInfo.f0,
        trainData, linearModelType, MLEnvironmentFactory.get(getMLEnvironmentId()));
    // Prepare the meta info of linear model.
    DataSet<Params> meta = labelInfo.f0
        .mapPartition(new CreateMeta(modelName, linearModelType, isRegProc, params))
        .setParallelism(1);
    // Build linear model rows, the format to be output.
    DataSet<Row> modelRows;
    String[] featureColTypes = getFeatureTypes(in, params.get(LinearTrainParams.FEATURE_COLS));
    modelRows = coefVectorSet
        .mapPartition(new BuildModelFromCoefs(labelInfo.f1,
            params.get(LinearTrainParams.FEATURE_COLS),
            params.get(LinearTrainParams.STANDARDIZATION),
            params.get(LinearTrainParams.WITH_INTERCEPT), featureColTypes))
        .withBroadcastSet(meta, META)
        .withBroadcastSet(statInfo.f1, MEAN_VAR)
        .setParallelism(1);
    // Convert the model rows to table.
    this.setOutput(modelRows, new LinearModelDataConverter(labelInfo.f1).getModelSchema());
    return (T)this;
}
```

我们后续还会对此逻辑进行细化。

0x04 基础功能

我们首先介绍下相关基础功能和相关概念，比如损失函数，目标函数，梯度等。

4.1 损失函数

损失函数涉及到若干概念。

4.1.1 导数和偏导数

导数也是函数，是函数的变化率与位置的关系。导数代表了在自变量变化趋于无穷小的时候，函数值的变化与自变量的变化的比值。几何意义是这个点的切线。物理意义是该时刻的（瞬时）变化率。

导数反映的是函数y=f(x)在某一点处沿x轴正方向的变化率。直观地看，也就是在x轴上某一点处，如果f′(x)>0，说明f(x)的函数值在x点沿x轴正方向是趋于增加的；如果f′(x)<0，说明f(x)的函数值在x点沿x轴正方向是趋于减少的。

一元导数表征的是：一元函数 f(x)与自变量 x 在某点附近变化的比率（变化率，斜率）。

如果是多元函数呢？则为偏导数。偏导数是多元函数“退化”成一元函数时的导数，这里“退化”的意思是固定其他变量的值，只保留一个变量，依次保留每个变量，则N元函数有N个偏导数。偏导数为函数在每个位置处沿着自变量坐标轴方向上的导数（切线斜率）。二元函数的偏导数表征的是：函数 F(x,y) 与自变量 x（或y） 在某点附近变化的比率（变化率）。

4.1.2 方向导数

导数和偏导数的定义中，均是沿坐标轴正方向讨论函数的变化率。那么当我们讨论函数沿任意方向的变化率时，也就引出了方向导数的定义，即：某一点在某一趋近方向上的导数值。

方向导数就是偏导数合成向量与方向向量的内积。方向导数的本质是一个数值，简单来说其定义为：一个函数沿指定方向的变化率。

4.1.3 Hessian矩阵

在一元函数求解的问题中，我们可以很愉快的使用牛顿法求驻点。但在机器学习的优化问题中，我们要优化的都是多元函数，x往往不是一个实数，而是一个向量，所以将牛顿求根法利用到机器学习中时，x 是一个向量, y 也是一个向量，对 x 求导以后得到的是一个矩阵，就是Hessian矩阵。

在数学中，海森矩阵（Hessian matrix 或 Hessian）是一个自变量为向量的实值函数的二阶偏导数组成的方块矩阵。多元函数的二阶导数就是一个海森矩阵。

4.1.4 平方损失函数 in Alink

前面提到，线性回归中损失函数就是平方损失函数。我们来看看实现。后续实现将调用此类的 loss 和 derivative，具体遇到时候再讲。

UnaryLossFunc是接口，代表一元损失函数。它定义的每个函数都有两个输入 (eta and y)，Alink把这两个输入的差作为损失函数的一元变量。基本API是求损失，求导数，求二阶导数。

```
public interface UnaryLossFunc extends Serializable {
        // Loss function.
        double loss(double eta, double y);
        // The derivative of loss function.
        double derivative(double eta, double y);
        // The second derivative of the loss function.
        double secondDerivative(double eta, double y);
}
```

平方损失函数具体实现如下：

```
public class SquareLossFunc implements UnaryLossFunc {

   @Override
   public double loss(double eta, double y) {
      return 0.5 * (eta - y) * (eta - y);
   }

   @Override
   public double derivative(double eta, double y) {
      return eta - y;
   }

   @Override
   public double secondDerivative(double eta, double y) {
      return 1;
   }
}
```

4.2 目标函数

这里涉及的概念是梯度，梯度下降法。

4.2.1 梯度

对于模型优化，我们要选择最优的 θ，使得 f(x) 最接近真实值。这个问题就转化为求解最优的 θ，使损失函数 J(θ) 取最小值。那么如何解决这个转化后的问题呢？这又牵扯到一个概念：**梯度下降（Radient Descent）**。

所以我们首先要温习下梯度。

- 向量的定义是有方向（direction）有大小（magnitude）的量。
- 梯度其实是一个向量，即有方向有大小；其定义为：一个多元函数对于其自变量分别求偏导数，这些偏导数所组成的向量就是函数的梯度。
- 梯度即函数在某一点最大的方向导数，函数沿梯度方向函数有最大的变化率。
- 梯度的第一层含义就是**"方向导数的最大值"**。
- 当前位置的**梯度方向**，为函数在该位置处**方向导数最大的方向**，也是函数值**上升最快的方向**，反方向为下降最快的方向；
- 梯度的几何含义就是：沿向量所在直线的方向变化率最大。

4.2.2 梯度下降法

梯度下降法是一个一阶最优化算法，它的核心思想是：要想最快找到一个函数的局部极小值，必须沿函数当前点对应"梯度"（或者近似梯度）的反方向（下降）进行规定步长"迭代"搜索。**沿梯度（斜率）的反方向移动**，这就是**"梯度下降法"**。

既然在变量空间的某一点处，函数沿梯度方向具有最大的变化率，那么在优化目标函数的时候，自然是沿着负梯度方向去减小函数值，以此达到我们的优化目标。

梯度下降中的下降，意思是让函数的未知数随着梯度的方向运动。什么是梯度的方向呢？把这一点带入到梯度函数中，结果为正，那我们就把这一点的值变小一些，同时就是让梯度变小些；当这一点带入梯度函数中的结果为负的时候，就给这一点的值增大一些。

如何沿着负梯度方向减小函数值呢？既然梯度是偏导数的集合，同时梯度和偏导数都是向量，那么参考向量运算法则，我们在每个变量轴上减小对应变量值即可。

梯度下降就是让梯度中所有偏导函数都下降到最低点的过程.(划重点:下降)。都下降到最低点了,那每个未知数(或者叫维度)的最优解就得到了，所以他是解决函数最优化问题的算法。

“最小二乘法”和“梯度下降法”，前者用于“搜索最小误差”，后者用于“用最快的速度搜索”，二者常常配合使用。对最小二乘法的参数调优就转变为了求这个二元函数的极值问题，也就是说可以应用“梯度下降法”了。

在最小二乘函数中，已拥有的条件是一些样本点和样本点的结果，就是矩阵X和每一条X样本的lable值y。X是矩阵，y是向量。所以我们要知道，梯度下降中求偏导数的未知数不是x和y，而是x的参数w。

4.2.3 目标函数 in Alink

目标函数的基类是OptimObjFunc，其提供API 比如计算梯度，损失，hessian矩阵，以及依据采样点更新梯度和hessian矩阵。 其几个派生类如下，从注释中可以看到使用范围。

我们可以看到正则化(regularization) L1, L2范数，这是相比损失函数增加的模块。

```
public abstract class OptimObjFunc implements Serializable {
    protected final double l1;
    protected final double l2; // 正则化(regularization) L1, L2范数。
    protected Params params;
  .....
}

// Unary loss object function.
public class UnaryLossObjFunc extends OptimObjFunc

// The OptimObjFunc for multilayer perceptron.
public class AnnObjFunc extends OptimObjFunc

// Accelerated failure time Regression object function.
public class AftRegObjFunc extends OptimObjFunc

// Softmax object function.
public class SoftmaxObjFunc extends OptimObjFunc
```

对于线性模型，BaseLinearModelTrainBatchOp 中会根据模型类型来生成目标函数，可以看到在生成目标函数同时，也相应设置了不同的损失函数，其中 SquareLossFunc 就是我们之前提到的。

```
public static OptimObjFunc getObjFunction(LinearModelType modelType, Params params) {
    OptimObjFunc objFunc;
    // For different model type, we must set corresponding loss object function.
    switch (modelType) {
        case LinearReg:
            // 我们这里!
            objFunc = new UnaryLossObjFunc(new SquareLossFunc(), params);
            break;
        case SVR:
            double svrTau = params.get(LinearSvrTrainParams.TAU);
            objFunc = new UnaryLossObjFunc(new SvrLossFunc(svrTau), params);
            break;
        case LR:
            objFunc = new UnaryLossObjFunc(new LogLossFunc(), params);
            break;
        case SVM:
            objFunc = new UnaryLossObjFunc(new SmoothHingeLossFunc(), params);
            break;
        case Perceptron:
            objFunc = new UnaryLossObjFunc(new PerceptronLossFunc(), params);
            break;
        case AFT:
            objFunc = new AftRegObjFunc(params);
            break;
        default:
            throw new RuntimeException("Not implemented yet!");
    }
    return objFunc;
}
```

4.2.4 一元目标函数 in Alink

一元目标函数就是我们线性回归用到的目标函数，其只有一个新增变量： unaryLossFunc。就是一元损失函数。

```
/**
 * Unary loss object function.
 */
public class UnaryLossObjFunc extends OptimObjFunc {
    private UnaryLossFunc unaryLossFunc;
}
```

一元目标函数提供了很多功能，我们这里用到主要是：

- calcGradient： 根据一组采样点计算梯度，这是从基类OptimObjFunc集成的。
- updateGradient： 根据一个采样点更新梯度；

- calcSearchValues ： 为线性搜索计算损失；

4.2.4.1 依据一组采样点计算梯度

对于本文，这里更新的是损失函数的梯度。

再次啰嗦下，损失函数用来度量拟合的程度，从而评估模型拟合的好坏，记为 ***J(θ)***。注意，损失函数是关于 **θ** 的函数！也就是说，对于损失函数来讲，**θ**不再是函数的参数，而是损失函数的自变量！

当我们计算损失时，是将每个样本中的特征 ***xi*** 和对应的目标变量真实值 ***yi*** 带入损失函数，此时，损失函数中就只剩下 **θ** 是未知的。

损失函数的梯度即对 **θi** 求偏导，由于损失函数是关于 **θ** 的函数，因此，**θ** 的取值不同，得出来的的梯度向量也是不同的。借用"下山"的比喻来解释，**θ** 的不同取值，相当于处于山上的不同位置，每一个位置都会计算出一个梯度向量▽J(θ)。

这里的 l1, l2 就是之前提到的正则化(regularization) L1, L2范数。

```
/**
 * Calculate gradient by a set of samples.
 *
 * @param labelVectors train data.
 * @param coefVector   coefficient of current time.
 * @param grad         gradient.
 * @return weight sum
 */
public double calcGradient(Iterable<Tuple3<Double, Double, Vector>> labelVectors,
                           DenseVector coefVector, DenseVector grad) {
    double weightSum = 0.0;
    for (int i = 0; i < grad.size(); i++) {
        grad.set(i, 0.0);
    }

// 对输入的样本集合labelVectors逐个计算梯度
    for (Tuple3<Double, Double, Vector> labelVector : labelVectors) {
        if (labelVector.f2 instanceof SparseVector) {
           ((SparseVector)(labelVector.f2)).setSize(coefVector.size());
        }

// 以这个样本为例
labelVector = {Tuple3@9895} "(1.0,16.8,1.0 1.0 1.4657097546055162 1.4770978917519928)"
 f0 = {Double@9903} 1.0
 f1 = {Double@9904} 16.8
 f2 = {DenseVector@9905} "1.0 1.0 1.4657097546055162 1.4770978917519928"

        weightSum += labelVector.f0; // labelVector.f0是权重
        updateGradient(labelVector, coefVector, grad);
    }
    if (weightSum > 0.0) {
        grad.scaleEqual(1.0 / weightSum);
    }
// l2正则化
    if (0.0 != this.l2) {
        grad.plusScaleEqual(coefVector, this.l2 * 2);
    }
// l1正则化
    if (0.0 != this.l1) {
        double[] coefArray = coefVector.getData();
        for (int i = 0; i < coefVector.size(); i++) {
            grad.add(i, Math.signum(coefArray[i]) * this.l1);
        }
    }
    return weightSum;
}
```

4.2.4.2 根据一个采样点更新梯度

这里 labelVector.f0是权重，labelVector.f1是 y，labelVector.f2是 x-vec 四维向量，coefVector是w系数向量。

- getEta是点积，即 x向量 与 当前w系数的点积，就是当前计算的 y。
- labelVector.f0 * unaryLossFunc.derivative(eta, labelVector.f1); 就是调用 SquareLossFunc.derivative 函数来计算一阶导数。
- updateGrad.plusScaleEqual(labelVector.f2, div); 就是在原有梯度基础上更新梯度

```
public class UnaryLossObjFunc extends OptimObjFunc {
    /**
     * Update gradient by one sample.
     *
     * @param labelVector a sample of train data.
     * @param coefVector  coefficient of current time.
     * @param updateGrad  gradient need to update.
     */
```

```
    @Override
    protected void updateGradient(Tuple3<Double, Double, Vector> labelVector, DenseVector coefV
ector, DenseVector updateGrad) {
        // 点积，就是当前计算出来的y
        double eta = getEta(labelVector, coefVector);
        // 一阶导数。labelVector.f0是权重
        double div = labelVector.f0 * unaryLossFunc.derivative(eta, labelVector.f1);
        // 点乘之后还需要相加。labelVector.f2 就是x—vec，比如 1.0 1.0 1.4657097546055162 1.477097891
7519928
        updateGrad.plusScaleEqual(labelVector.f2, div);
    }

    private double getEta(Tuple3<Double, Double, Vector> labelVector, DenseVector coefVector) {
        // 点积，表示第 i 次迭代中节点上的第 k 个特征向量与特征权重分量的点乘。coefVector中第 c 项表示为第 i
 次迭代中特征权重向量在第 c 列节点上的分量
        return MatVecOp.dot(labelVector.f2, coefVector);
    }
}

/**
* Plus with another vector scaled by "alpha".
*/
public void plusScaleEqual(Vector other, double alpha) {
        if (other instanceof DenseVector) {
                BLAS.axpy(alpha, (DenseVector) other, this);
        } else {
                BLAS.axpy(alpha, (SparseVector) other, this);
        }
}
```

4.3 优化函数

Alink中提供给了一系列并行优化函数，比如GD, SGD, LBFGS, OWLQN, NEWTON method。

其基类是Optimizer。

```
public abstract class Optimizer {
    protected final DataSet<?> objFuncSet; // 具体目标函数，计算梯度和损失
    protected final DataSet<Tuple3<Double, Double, Vector>> trainData; //训练数据
    protected final Params params; //参数
    protected DataSet<Integer> coefDim; //dimension of features.
    protected DataSet<DenseVector> coefVec = null; //最终系数w
    .......
}
```

线性回归主要用到了LBFGS算法。

```
public class Lbfgs extends Optimizer
```

具体调用如下

```
public static DataSet<Tuple2<DenseVector, double[]>> optimize(.....) {
    // Loss object function
    DataSet<OptimObjFunc> objFunc = session.getExecutionEnvironment()
        .fromElements(getObjFunction(modelType, params));

    if (params.contains(LinearTrainParams.OPTIM_METHOD)) {
        LinearTrainParams.OptimMethod method = params.get(LinearTrainParams.OPTIM_METHOD);
        return OptimizerFactory.create(objFunc, trainData, coefficientDim, params, method).opti
mize();
    } else if (params.get(HasL1.L_1) > 0) {
        return new Owlqn(objFunc, trainData, coefficientDim, params).optimize();
    } else {
        // 我们的程序将运行到这里
        return new Lbfgs(objFunc, trainData, coefficientDim, params).optimize();
    }
}
```

机器学习基本优化套路是：

```
准备数据 ----> 优化函数 ----> 目标函数 ----> 损失函数
```

对应我们这里是

```
BaseLinearModelTrainBatchOp.linkFrom(整体逻辑) -----> Lbfgs(继承Optimizer) ----> UnaryLossObjFunc
(继承OptimObjFunc) ----> SquareLossFunc(继承UnaryLossFunc)
```

0x05 数据准备

看完完底层功能，我们再次回到线性回归总体流程。

总结 BaseLinearModelTrainBatchOp.linkFrom 的基本流程如下：(发现某些媒体对于列表排版支持不好，所以加上序号)。

首先再给出输入一个例子：`Row.of("$3$0:1.0 1:7.0 2:9.0", "1.0 7.0 9.0", 1.0, 7.0, 9.0, 16.8),` 这里后面 4 项对应列名是 `"f0", "f1", "f2", "label"` 。

- 1)获取到label的信息，包括label数值和种类。 labelInfo = getLabelInfo() 这里有一个 distinct 操作，所以会去重。最后得到label的可能取值范围 ： 0，1，类型是 Double。
- 2)用transform函数把输入转换成三元组Tuple3<weight, label, feature vector>。具体说，会把输入中的三个特征"f0", "f1", "f2" 转换为一个向量 vec, 我们以后称之为x-vec。重点就在于特征变成了一个向量。所以这个三元组可以认为是 **<权重, y-value, x-vec>**。
- 3)用statInfo = getStatInfo() 获取统计变量，包括vector size, mean和variance。这里流程比较复杂。
  - 3.1)用trainData.map{return value.f2;}来获取训练数据中的 x-vec。
  - 3.2)调用StatisticsHelper.summary来对 x-vec 做处理
    - 3.2.1)调用 summarizer
      - 3.2.1.1)调用 mapPartition(new VectorSummarizerPartition(bCov))
        - 3.2.1.1.1)调用VectorSummarizerPartition.mapPartition，其遍历列表，列表中的每一个变量 sv 是 x-vec。srt = srt.visit(sv)，会根据每一个新输入重新计算count, sum，squareSum，normL1..，这样就得到了本partiton中输入每列的这些统计数值。
      - 3.2.1.2)调用 reduce(VectorSummarizerUtil.merge(value1, value2)) 来归并每一个partition的结果。
    - 3.2.2)调用map(BaseVectorSummarizer summarizer)，其实调用到DenseVectorSummarizer，就是生成一个DenseVectorSummary向量，里面是count，sum，squareSum，normL1，min，max，numNonZero。
  - 3.3)调用 coefficientDim = summary.map
  - 3.4)调用 meanVar = coefficientDim.map，最后得到 Tuple2.of(coefficientDim, meanVar)
- 4)preProcess(initData, params, statInfo.f1) 用3) 计算的结果 对输入数据做标准化和插值 standardization and interception。上面得到的 meanVar 将会作为参数传入。这里是对 x-vec 做标准化。比如原始输入Row是"(1.0,16.8,1.0 7.0 9.0)"，其中 x-vec 是"1.0 7.0 9.0"，进行标准化之后，x-vec 变成了 4 项 ： { 第1项是固定值 "1.0 "， 所以4 项 是 "1.0 1.0 1.4657097546055162 1.4770978917519928" }，所以转换后的Row是"(1.0,16.8,1.0 1.0 1.4657097546055162 1.4770978917519928)"。即weight 是1.0，y-value是16.8，后续4个是x-vec。
- 以上完成了对数据的处理。
- 5)调用 optimize(params, statInfo.f0, trainData, linearModelType) 通过对损失函数求最小值从而对模型优化。(使用L-BFGS算法，会单独拿出来讲解)
- 6)调用 mapPartition(new CreateMeta()) 来准备模型元数据。
- 7)调用 mapPartition(new BuildModelFromCoefs) 来建立模型。

可以看到，数据准备占据了很大部分，下面我们看看数据准备的几个步骤。

5.1 获取label信息

此处代码对应上面基本流程的 1)

因为之前有一个distinct操作，所以会去重。最后得到label的可能取值范围 ： 0，1，类型是 Double。

```
private Tuple2<DataSet<Object>, TypeInformation> getLabelInfo(BatchOperator in,
                                                              Params params,
                                                              boolean isRegProc) {
        String labelName = params.get(LinearTrainParams.LABEL_COL);
        // Prepare label values
        DataSet<Object> labelValues;
        TypeInformation<?> labelType = null;
        if (isRegProc) {
            // 因为是回归，所以是这里
            labelType = Types.DOUBLE;
            labelValues = MLEnvironmentFactory.get(in.getMLEnvironmentId())
                .getExecutionEnvironment().fromElements(new Object());
        } else {
          .....
        }
        return Tuple2.of(labelValues, labelType);
}
```

5.2 把输入转换成三元组

此处代码对应上面基本流程的 2) 。

用transform函数把输入转换成三元组Tuple3<weight, label, feature vector>。具体说，会把输入中的三个特征"f0", "f1", "f2" 转换为一个向量 vec, 我们以后称之为x-vec。重点就在于特征变成了一个向量。所以这个三元组可以认为是 **<权重, y-value, x-vec>**。

```
private DataSet<Tuple3<Double, Double, Vector>> transform(BatchOperator in,
                                                          Params params,
```

```
                                                    DataSet<Object> labelValues,
                                                    boolean isRegProc) {
    ......
    // 获取Schema
    TableSchema dataSchema = in.getSchema();
    // 获取各种index
    int labelIdx = TableUtil.findColIndexWithAssertAndHint(dataSchema.getFieldNames(), labelNam
e);
    ......
    int weightIdx = weightColName != null ? TableUtil.findColIndexWithAssertAndHint(in.getColNa
mes(), weightColName) : -1;
    int vecIdx = vectorColName != null ? TableUtil.findColIndexWithAssertAndHint(in.getColNames
(), vectorColName) : -1;
    // 用transform函数把输入转换成三元组Tuple3<weight, label, feature vector>
    return in.getDataSet().map(new Transform(isRegProc, weightIdx, vecIdx, featureIndices, labe
lIdx)).withBroadcastSet(labelValues, LABEL_VALUES);
}
```

这里对应的变量打印出来为

```
params = {Params@2745} "Params {featureCols=["f0","f1","f2"], labelCol="label", predictionCol="
linpred"}"
labelValues = {DataSource@2845}
isRegProc = true
featureColNames = {String[3]@2864}
 0 = "f0"
 1 = "f1"
 2 = "f2"
labelName = "label"
weightColName = null
vectorColName = null
dataSchema = {TableSchema@2866} "root\n |-- svec: STRING\n |-- vec: STRING\n |-- f0: DOUBLE\n |
-- f1: DOUBLE\n |-- f2: DOUBLE\n |-- label: DOUBLE\n"
featureIndices = {int[3]@2878}
 0 = 2
 1 = 3
 2 = 4
labelIdx = 5
weightIdx = -1
vecIdx = -1
```

具体在runtime时候，会进入到Transform.map函数。我们可以看到，会把输入中的三个特征"f0", "f1", "f2", 转换为一个向量 vec, 我们以后称之为x-vec。

```
private static class Transform extends RichMapFunction<Row, Tuple3<Double, Double, Vector>> {
    @Override
    public Tuple3<Double, Double, Vector> map(Row row) throws Exception {
        // 获取权重
        Double weight = weightIdx != -1 ? ((Number)row.getField(weightIdx)).doubleValue() : 1.0
;
        // 获取label
        Double val = FeatureLabelUtil.getLabelValue(row, this.isRegProc,
            labelIdx, this.positiveLableValueString);
        if (featureIndices != null) {
            // 获取x-vec
            DenseVector vec = new DenseVector(featureIndices.length);
            for (int i = 0; i < featureIndices.length; ++i) {
                vec.set(i, ((Number)row.getField(featureIndices[i])).doubleValue());
            }
            // 构建三元组
            return Tuple3.of(weight, val, vec);
        } else {
            Vector vec = VectorUtil.getVector(row.getField(vecIdx));
            return Tuple3.of(weight, val, vec);
        }
    }
}
```

如果对应原始输入 `Row.of("$3$0:1.0 1:7.0 2:9.0", "1.0 7.0 9.0", 1.0, 7.0, 9.0, 16.8),` ，则程序中各种变量为：

```
row = {Row@9723} "$3$0:1.0 1:7.0 2:9.0,1.0 7.0 9.0,1.0,7.0,9.0,16.8"
weight = {Double@9724} 1.0
val = {Double@9725} 16.8
vec = {DenseVector@9729} "1.0 7.0 9.0"
vecIdx = -1
featureIndices = {int[3]@9726}
 0 = 2
 1 = 3
 2 = 4
```

5.3 获取统计变量

用getStatInfo() 对输入数据做标准化和插值 standardization and interception。

此处代码对应上面基本流程的 3)

3. 用statInfo = getStatInfo() 获取统计变量，包括vector size, mean和variance。这里流程比较复杂。

- 3.1)用trainData.map{return value.f2;}来获取训练数据中的 x-vec。
- 3.2)调用StatisticsHelper.summary来对 x-vec 做处理
  - 3.2.1)调用 summarizer
    - 3.2.1.1)调用 mapPartition(new VectorSummarizerPartition(bCov))
      - 3.2.1.1.1)调用VectorSummarizerPartition.mapPartition，其遍历列表，列表中的每一个变量 sv 是 x-vec。srt = srt.visit(sv)，会根据每一个新输入重新计算count，sum，squareSum，normL1..，这样就得到了本partiton中输入每列的这些统计数值。
    - 3.2.1.2)调用 reduce(VectorSummarizerUtil.merge(value1, value2)) 来归并每一个partition的结果。
  - 3.2.2)调用map(BaseVectorSummarizer summarizer)，其实调用到DenseVectorSummarizer，就是生成一个DenseVectorSummary向量，里面是count，sum，squareSum，normL1，min，max，numNonZero。
- 3.3)调用 coefficientDim = summary.map
- 3.4)调用 meanVar = coefficientDim.map，最后得到 Tuple2.of(coefficientDim, meanVar)

```
private Tuple2<DataSet<Integer>, DataSet<DenseVector[]>> getStatInfo(
    DataSet<Tuple3<Double, Double, Vector>> trainData, final boolean standardization) {
    if (standardization) {
        DataSet<BaseVectorSummary> summary = StatisticsHelper.summary(trainData.map(
            new MapFunction<Tuple3<Double, Double, Vector>, Vector>() {
                @Override
                public Vector map(Tuple3<Double, Double, Vector> value) throws Exception {
                    return value.f2; //获取训练数据中的 x-vec
                }
            }).withForwardedFields());

        DataSet<Integer> coefficientDim = summary.map(new MapFunction<BaseVectorSummary, Intege
r>() {
            public Integer map(BaseVectorSummary value) throws Exception {
                return value.vectorSize(); // 获取dimension
            }
        });

        DataSet<DenseVector[]> meanVar = summary.map(new MapFunction<BaseVectorSummary, DenseVe
ctor[]>() {
            public DenseVector[] map(BaseVectorSummary value) {
                if (value instanceof SparseVectorSummary) {
                    // 计算min, max
                    DenseVector max = ((SparseVector)value.max()).toDenseVector();
                    DenseVector min = ((SparseVector)value.min()).toDenseVector();
                    for (int i = 0; i < max.size(); ++i) {
                        max.set(i, Math.max(Math.abs(max.get(i)), Math.abs(min.get(i))));
                        min.set(i, 0.0);
                    }
                    return new DenseVector[] {min, max};
                } else {
                    // 计算standardDeviation
                    return new DenseVector[] {(DenseVector)value.mean(),
                        (DenseVector)value.standardDeviation()};
                }
            }
        });
        return Tuple2.of(coefficientDim, meanVar);
    }
}
```

5.4 对输入数据做标准化和插值

这里对应基本流程的 4) 。

对输入数据做标准化和插值 standardization and interception。上面得到的 meanVar 作为参数传入。这里是对 x-vec 做标准化。

比如原始输入Row是 `"(1.0,16.8,1.0 7.0 9.0)"` ，其中 x-vec 是 `"1.0 7.0 9.0"` ，进行标准化之后，x-vec 变成了 4 项，第一项是固定值 "1.0 "， 4 项 是 `"1.0 1.0 1.4657097546055162 1.4770978917519928"` ，所以转换后的Row是 `"(1.0,16.8,1.0 1.0 1.4657097546055162 1.4770978917519928)"` 。

为什么第一项是固定值 "1.0 " ？因为按照线性模型 `f(x)=w^Tx+b` ，我们应该得出一个常数 b，这里设定 "1.0 "，就是 b 的初始值。

```
private DataSet<Tuple3<Double, Double, Vector>> preProcess(
    return initData.map(
```

```
        new RichMapFunction<Tuple3<Double, Double, Vector>, Tuple3<Double, Double, Vector>>() {
            private DenseVector[] meanVar;

            @Override
            public Tuple3<Double, Double, Vector> map(Tuple3<Double, Double, Vector> value){
// value = {Tuple3@9791} "(1.0,16.8,1.0 7.0 9.0)"
                Vector aVector = value.f2;
// aVector = {DenseVector@9792} "1.0 7.0 9.0"
                if (aVector instanceof DenseVector) {
                    DenseVector bVector;
                    if (standardization) {
                        if (hasInterceptItem) {
                            bVector = new DenseVector(aVector.size() + 1);
                            bVector.set(0, 1.0); // 设定了固定值
                            for (int i = 0; i < aVector.size(); ++i) {
                                // 对输入数据做标准化和插值
                                bVector.set(i + 1, (aVector.get(i) - meanVar[0].get(i)) / meanV
ar[1].get(i));
                            }
                        }
                    }
// bVector = {DenseVector@9814} "1.0 1.0 1.4657097546055162 1.4770978917519928"
                    return Tuple3.of(value.f0, value.f1, bVector);
                }
            }
        }).withBroadcastSet(meanVar, MEAN_VAR);
}

// 这里是对 x-vec 做标准化。比如原始输入Row是"(1.0,16.8,1.0 7.0 9.0)"，其中 x-vec 是"1.0 7.0 9.0"，进行标准化之后，x-vec 变成了 4 项，第一项是 "1.0 "，是 "1.0 1.0 1.4657097546055162 1.4770978917519928"，所以转换后的Row是"(1.0,16.8,1.0 1.0 1.4657097546055162 1.4770978917519928)"
```

至此，输入处理完毕。

比如原始输入Row是"(1.0,16.8,1.0 7.0 9.0)"，其中 x-vec 是"1.0 7.0 9.0"。

进行标准化之后，x-vec 变成了 4 项：{ 第1项是固定值 "1.0 "， 所以4 项 是 "1.0 1.0 1.4657097546055162 1.4770978917519928" }，

转换后的Row是"(1.0,16.8,1.0 1.0 1.4657097546055162 1.4770978917519928)"。即weight 是1.0，y-value是16.8，后续4个是x-vec。

下面我们可以开始进行优化模型了，敬请期待下文。

0xFF 参考

[终于理解了方向导数与梯度](#)

[导数，方向导数，梯度（Gradient）与梯度下降法（Gradient Descent）的介绍（非原创）](#)

[梯度向量与梯度下降法](#)

[直观理解梯度，以及偏导数、方向导数和法向量等](#)

[梯度（Gradient）与梯度下降法（Gradient Descent）](#)

[梯度与梯度下降法](#)

[梯度下降算法过程详细解读](#)

https://www.zhihu.com/question/25627482/answer/321719657)

[Hessian矩阵以及在图像中的应用](#)

https://blog.csdn.net/weixin_39445556/article/details/84502260)

[《分布式机器学习算法、理论与实践_刘铁岩》](#)

https://zhuanlan.zhihu.com/p/29672873)

https://www.zhihu.com/question/36425542

https://zhuanlan.zhihu.com/p/32821110)

https://blog.csdn.net/hei653779919/article/details/106409818)

[CRF L-BFGS Line Search原理及代码分析](#)

[步长与学习率](#)

https://blog.csdn.net/IMWTJ123/article/details/88709023)

[线性回归、梯度下降（Linear Regression、Gradient Descent）](#)

[机器学习系列（三）——目标函数和损失函数](#)