Alink漫谈(十一)： 线性回归 之 L-BFGS优化

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。本文介绍了线性回归的L-BFGS优化在Alink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。

本系列目前已有十一篇，欢迎大家指点

Alink漫谈(一)：从KMeans算法实现不同看Alink设计思想

Alink漫谈(二)：从源码看机器学习平台Alink设计和架构

[Alink漫谈之三] AllReduce通信模型

Alink漫谈(四)：模型的来龙去脉

Alink漫谈(五)：迭代计算和Superstep

Alink漫谈(六)：TF-IDF算法的实现

Alink漫谈(七)：如何划分训练数据集和测试数据集

Alink漫谈(八)：二分类评估 AUC、K-S、PRC、Precision、Recall、LiftChart 如何实现

Alink漫谈(九)： 特征工程 之 特征哈希/标准化缩放

Alink漫谈(十)： 线性回归实现 之 数据预处理

0x01 回顾

到目前为止，已经处理完毕输入，接下来就是优化。优化的主要目标是找到一个方向，参数朝这个方向移动之后使得损失函数的值能够减小，这个方向往往由一阶偏导或者二阶偏导各种组合求得。 所以我们再次复习下基本思路。

1.1 优化基本思路

对于线性回归模型 f(x) = w'x+e，我们构造一个Cost函数（损失函数）**J(θ)**，并且通过找到 **J(θ)** 函数的最小值，就可以确定线性模型的系数 w 了。

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**。

我们要做的是依据我们的训练集，选取最优的θ，在我们的训练集中让f(x)尽可能接近真实的值。我们定义了一个函数来描述 "f(x)和真实的值y之间的差距"，这个函数称为目标函数，表达式如下：

$$J(\theta) \approx \frac{1}{2} \sum_{i \approx 1}^{m} (f_\theta(x^{(i)} — y^{(i)})^2$$

$$J(\theta) \approx \frac{1}{2} \sum_{i \approx 1}^{m} (f_\theta(x^{(i)} — y^{(i)})^2$$

这里的目标函数就是著名的**最小二乘函数**。

我们要选择最优的θ，使得f(x)最近进真实值。这个问题就转化为求解最优的θ，使目标函数 J(θ) 取最小值。

1.2 各类优化方法

寻找合适的*W*令目标函数*f(W)* 最小，是一个无约束最优化问题，解决这个问题的通用做法是随机给定一个初始的*W0*，通过迭代，在每次迭代中计算目标函数的下降方向并更新*W*，直到目标函数稳定在最小的点。

不同的优化算法的区别就在于目标函数下降方向*Dt*的计算。下降方向是通过对目标函数在当前的*W*下求一阶倒数（梯度，Gradient）和求二阶导数（海森矩阵，Hessian Matrix）得到。常见的算法有梯度下降法、牛顿法、拟牛顿法。

- 梯度下降法直接采用目标函数在当前*W*的梯度的反方向作为下降方向。
- 牛顿法是在当前*W*下，利用二次泰勒展开近似目标函数，然后利用该近似函数来求解目标函数的下降方向。其中*Bt*为目标函数*f(W)*在*Wt*处的海森矩阵。这个搜索方向也称作牛顿方向。
- 拟牛顿法只要求每一步迭代中计算目标函数的梯度，通过拟合的方式找到一个近似的海森矩阵用于计算牛顿方向。
- L-BFGS(Limited-memory BFGS)则是解决了BFGS中每次迭代后都需要保存*N*N*阶海森逆矩阵的问题，只需要保存每次迭代的两组向量和一组标量即可。

Alink中，UnaryLossObjFunc是目标函数，SquareLossFunc 是损失函数，使用L-BFGS算法对模型进行优化。

即 优化方向由**拟牛顿法L-BFGS**搞定(具体如何弄就是看f(x)的泰勒二阶展开)，损失函数最小值是**平方损失函数**来计算。

0x02 基本概念

因为L-BFGS算法是拟牛顿法的一种，所以我们先从牛顿法的本质泰勒展开开始介绍。

2.1 泰勒展开

泰勒展开是希望基于某区间一点x_0展开，用一组简单的幂函数来近似一个复杂的函数f(x)在该区间的局部。泰勒展开的应用场景例如：我们很难直接求得sin(1)的值，但我们可以通过sin的泰勒级数求得sin(1)的近似值，且展开项越多，精度越高。计算机一般都是把 sin(x)进行泰勒展开进行计算的。

泰勒当年为什么要发明这条公式?

因为当时数学界对简单函数的研究和应用已经趋于成熟，而复杂函数，比如：f(x) = sin(x)ln(1+x) 这种一看就头疼的函数，还有那种根本就找不到表达式的曲线。除了代入一个x可以得到它的y，就啥事都很难干了。所以泰勒同学就迎难而上！决定让这些式子统统现出原形，统统变简单。

要让一个复杂函数变简单，能不能把它转换成别的表达式？泰勒公式一句话描述：就是用多项式函数去逼近光滑函数。即，根据"以直代曲、化整为零"的数学思想，产生了泰勒公式。

泰勒公式**通过把【任意函数表达式】转换（重写）为【多项式】形式**，是一种极其强大的函数**近似工具**。为什么说它强大呢?

- 多项式非常【友好】，三易，易计算，易求导，易积分
- 几何感觉和计算感觉都很直观，如抛物线和几次方就是底数自己乘自己乘几次

如何通俗推理?

泰勒公式干的事情就是：**使用多项式表达式估计（近似） 在 附近的值**。

当我们想要仿造一个东西的时候，无形之中都会按照如下思路，即先保证大体上相似，再保证局部相似，再保证细节相似，再保证更细微的地方相似……不断地细化下去，无穷次细化以后，仿造的东西将无限接近真品。真假难辨。

物理学家得出结论：把生活中关于"仿造"的经验运用到运动学问题中，如果想仿造一段曲线，那么首先应该保证曲线的起始点一样，其次保证起始点处位移随时间的变化率一样（速度相同），再次应该保证前两者相等的同时关于时间的二阶变化率一样（加速度相同）……如果随时间每一阶变化率（每一阶导数）都一样，那这俩曲线肯定是完全等价的。

所以如果泰勒想一个办法让自己避免接触 sin(x)这类函数，即**把这类函数替换掉。** 就可以根据这类函数的图像，仿造一个图像，与原来的图像相类似，这种行为在数学上叫近似。不扯这个名词。讲讲如何仿造图像。

仿造的第一步，就是让仿造的曲线也过这个点。

完成了仿造的第一步，很粗糙，甚至完全看不出来这俩有什么相似的地方，那就继续细节化。开始考虑曲线的变化趋势，即导数，保证在此处的导数相等。

经历了第二步，现在起始点相同了，整体变化趋势相近了，可能看起来有那么点意思了。想进一步精确化，应该考虑凹凸性。高中学过：表征图像的凹凸性的参数为"导数的导数"。所以，下一步就让二者的导数的导数相等。

起始点相同，增减性相同，凹凸性相同后，仿造的函数更像了。如果再继续细化下去，应该会无限接近。所以泰勒认为"**仿造一段曲线，要先保证起点相同，再保证在此处导数相同，继续保证在此处的导数的导数相同.....**"

泰勒展开式就是把一个三角函数或者指数函数或者其他比较难缠的函数用多项式替换掉。

也就是说，有一个**原函数f(x)** ，我再造一个图像与原函数图像相似的**多项式函数 g(x)** ，为了保证相似，我只需要保证这俩函数在某一点的**初始值相等，1阶导数相等，2阶导数相等，.....n阶导数相等。**

2.2 牛顿法

牛顿法的基本思路是，**在现有极小点估计值的附近对 f(x) 做二阶泰勒展开，进而找到极小点的下一个估计值。**其核心思想是利用迭代点 x_k 处的一阶导数(梯度)和二阶导数(Hessen矩阵)对目标函数进行二次函数近似，然后把二次模型的极小点作为新的迭代点，并不断重复这一过程，直至求得满足精度的近似极小值。

梯度下降算法是将函数在 xn 位置进行一次函数近似，也就是一条直线。计算梯度，从而决定下一步优化的方向是梯度的反方向。而牛顿法是将函数在 xn 位置进行二阶函数近似，也就是二次曲线。计算梯度和二阶导数，从而决定下一步的优化方向。

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f(x) 的一阶导数也是一个向量，再求导的二阶导数是一个矩阵，就是Hessian矩阵。

2.2.1 泰勒一阶展开

牛顿法求根可以按照泰勒一阶展开。例如对于方程 f(x) = 0，我们在任意一点 x0 处，进行一阶泰勒展开：

$$f(x) = f(x_0) + (x - x_0)f'(x_0)$$
$$f(x) = f(x_0) + (x - x_0)f'(x_0)$$

令 f(x) = 0，带入上式，即可得到：

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$
$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

注意，这里使用了近似，得到的 x 并不是方程的根，只是近似解。但是，x 比 x0 更接近于方程的根。

然后，利用迭代方法求解，以 x 为 x0，求解下一个距离方程的根更近的位置。迭代公式可以写成：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

2.2.2 泰勒二阶展开

机器学习、深度学习中，损失函数的优化问题一般是基于一阶导数梯度下降的。现在，从另一个角度来看，想要让损失函数最小化，这其实是一个最值问题，对应函数的一阶导数 f'(x) = 0。也就是说，如果我们找到了能让 f'(x) = 0 的点 x，损失函数取得最小值，也就实现了模型优化目标。

现在的目标是计算 f′(x) = 0 对应的 x，即 f′(x) = 0 的根。转化为求根问题，就可以利用上一节的牛顿法了。

与上一节有所不同，首先，对 f(x) 在 x0 处进行二阶泰勒展开：

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0)$$
$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0)$$

上式成立的条件是 f(x) 近似等于 f(x0)。令 f(x) = f(x0)，并对 (x - x0) 求导，可得：

$$x = x_0 - \frac{f'(x_0)}{f''(x_0)}$$
$$x = x_0 - \frac{f'(x_0)}{f''(x_0)}$$

同样，虽然 x 并不是最优解点，但是 x 比 x0 更接近 f'(x) = 0 的根。这样，就能得到最优化的迭代公式：

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

通过迭代公式，就能不断地找到 f'(x) = 0 的近似根，从而也就实现了损失函数最小化的优化目标。

2.2.3 高维空间

在机器学习的优化问题中，我们要优化的都是多元函数，所以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，f(x) 的一阶导数也是一个向量，再求导的二阶导数是一个矩阵，就是Hessian矩阵。

在高维空间，我们用梯度替代一阶导数，用Hessian矩阵替代二阶导数，牛顿法的迭代公式不变：

$$x_{k+1} = x_k - [Hf(x_k)^{-1}].J_f(x_k)$$
$$x_{k+1} = x_k - [Hf(x_k)^{-1}].J_f(x_k)$$

其中 J 定义为 **雅克比矩阵，对应一阶偏导数。**

推导如下：

我们假设f(x)是二阶可微实函数，把f(x)在xk处Taylor展开并取二阶近似为

$$f(x) \approx f(x^k) + \nabla f(x^k)T(x - x^k) + \frac{1}{2}(x - x^k)^T \nabla^2 f(x^k)(x - x^k)$$

x^k为当前的极小值估计值

$\nabla f(x^k)$是$f(x)$在x^k处的一阶导数

$\nabla^2 f(x)$是$f(x)$在x^k处的$Hessen$矩阵。

$$f(x) \approx f(x^k) + \nabla f(x^k)T(x - x^k) + \frac{1}{2}(x - x^k)^T \nabla^2 f(x^k)(x - x^k)$$

x^k 为当前的极小值估计值

$\nabla f(x^k)$ 是 $f(x)$ 在 x^k 处的一阶导数

$\nabla^2 f(x)$ 是 $f(x)$ 在 x^k 处的 $Hessen$ 矩阵。

我们的目标是求f(x)的最小值，而导数为0的点极有可能为极值点，故在此对f(x)求导，并令其导数为0，即∇f(x)=0，可得

$$\nabla f(x) = \nabla f(x^k) + \nabla^2 f(x^k)(x - x^k) = 0$$
$$\nabla f(x) = \nabla f(x^k) + \nabla^2 f(x^k)(x - x^k) = 0$$

设∇2f(x)可逆，由(2)可以得到牛顿法的迭代公式

$$x^{k+1} = x^k - \nabla^2 f(x^k)^{-1} \nabla f(x^k)$$
$$d = -\nabla^2 f(x^k)^{-1} \nabla f(x^k)$$被称为牛顿方向
$$x^{k+1} = x^k - \nabla^2 f(x^k)^{-1} \nabla f(x^k)$$
$$d = -\nabla^2 f(x^k)^{-1} \nabla f(x^k)$$ 被称为牛顿方向

当原函数存在一阶二阶连续可导时，可以采用牛顿法进行一维搜索，收敛速度快，具有局部二阶收敛速度。

2.2.4 牛顿法基本流程

总结(模仿)一下使用牛顿法求根的步骤：

a.已知函数的情况下,随机产生点.

b.由已知的点按照的公式进行k次迭代.

c.如果与上一次的迭代结果相同或者相差结果小于一个阈值时,本次结果就是函数的根.

伪代码如下

```
def newton(feature, label, iterMax, sigma, delta):
    '''牛顿法
    input:  feature(mat):特征
            label(mat):标签
            iterMax(int):最大迭代次数
            sigma(float), delta(float):牛顿法中的参数
    output: w(mat):回归系数
    '''
    n = np.shape(feature)[1]
    w = np.mat(np.zeros((n, 1)))
    it = 0
    while it <= iterMax:
        g = first_derivativ(feature, label, w)  # 一阶导数
        G = second_derivative(feature)  # 二阶导数
        d = -G.I * g
        m = get_min_m(feature, label, sigma, delta, d, w, g)  # 得到步长中最小的值m
        w = w + pow(sigma, m) * d
        it += 1
    return w
```

2.2.5 问题点及解决

牛顿法不仅需要计算 Hessian 矩阵，而且需要计算 Hessian 矩阵的逆。当数据量比较少的时候，运算速度不会受到大的影响。但是，当数据量很大，特别在深度神经网络中，计算 Hessian 矩阵和它的逆矩阵是非常耗时的。从整体效果来看，牛顿法优化速度没有梯度下降算法那么快。所以，目前神经网络损失函数的优化策略大多都是基于梯度下降。

另一个问题是，如果某个点的Hessian矩阵接近奇异（条件数过大），逆矩阵会导致数值不稳定，甚至迭代可能不会收敛。

当x的维度特别多的时候，我们想求得f(x) 的二阶导数很困难，而牛顿法求驻点又是一个迭代算法，所以这个困难我们还要面临无限多次，导致了牛顿法求驻点在机器学习中无法使用。有没有什么解决办法呢?

实际应用中，我们通常不去求解逆矩阵，而是考虑求解Hessian矩阵的近似矩阵，最好是只利用一阶导数近似地得到二阶导数的信息，从而在较少的计算量下得到接近二阶的收敛速率。这就是拟牛顿法。

拟牛顿法的想法其实很简单，就像是函数值的两点之差可以逼近导数一样，一阶导数的两点之差也可以逼近二阶导数。几何意义是求一阶导数的"割线"，取极限时，割线会逼近切线，矩阵B就会逼近Hessian矩阵。

拟牛顿法，同样可以顾名思义，就是模拟牛顿法，用一个近似于$\nabla^2 f(x)^{-1}$的矩阵H_{k+1}来替代$\nabla^2 f(x)^{-1}$

拟 牛 顿 法 ， 同 样 可 以 顾 名 思 义 ， 就 是 模 拟 牛 顿 法 ， 用 一 个 近 似 于 $\nabla^2 f(x)^{-1}$ 的 矩 阵 H_{k+1} 来 替

2.3 拟牛顿法

拟牛顿法就是模拟出Hessen矩阵的构造过程，通过迭代来逼近。主要包括DFP拟牛顿法，BFGS拟牛顿法。拟牛顿法只要求每一步迭代时知道目标函数的梯度。

各种拟牛顿法使用迭代法分别近似海森矩阵的逆和它自身；

在各种拟牛顿法中，一般的构造Hk+1的策略是，H1通常选择任意的一个n阶对称正定矩阵(一般为I)，然后通过不断的修正Hk给出Hk+1，即

$$H_{k+1} = H_k + \Delta H_k$$

ΔH_k称为校正矩阵

$H_{k+1} = H_k + \Delta H_k$ ΔH_k 称 为 校 正 矩 阵

比如：BFGS法每次更新矩阵H(Hessian矩阵的逆矩阵)需要的是第k步的迭代点差s和梯度差y，第k+1步的H相当于需要从开始到第k步的所用s和y的值。

我们要通过牛顿求驻点法和BFGS算法来求得一个函数的根，两个算法都需要迭代，所以我们干脆让他俩一起迭代就好了。两个算法都是慢慢逼近函数根，所以经过k次迭代以后，所得到的解就是机器学习中目标函数导函数的根。这种两个算法共同迭代的计算方式，我们称之为On The Fly.

在BFGS算法迭代的第一步，单位矩阵与梯度 g 相乘，就等于梯度 g，形式上同梯度下降的表达式是相同的。所以BFGS算法可以理解为从梯度下降逐步转换为牛顿法求函数解的一个算法。

虽然我们使用了BFGS算法来利用单位矩阵逐步逼近H矩阵，但是每次计算的时候都要存储D矩阵，D矩阵有多大。呢。假设我们的数据集有十万个维度(不算特别大)，那么每次迭代所要存储D矩阵的结果是74.5GB。我们无法保存如此巨大的矩阵内容，如何解决呢？使用L-BFGS算法.

2.4 L-BFGS算法

L-BFGS算法的基本思想是：算法只保存并利用最近m次迭代的曲率信息来构造海森矩阵的近似矩阵。

我们要通过牛顿求驻点法和BFGS算法来求得一个函数的根，两个算法都需要迭代，所以我们干脆让他俩一起迭代就好了。两个算法都是慢慢逼近函数根，所以经过k次迭代以后，所得到的解就是机器学习中目标函数导函数的根。这种两个算法共同迭代的计算方式，我们称之为On The Fly.

在BFGS算法迭代的第一步，单位矩阵与梯度g相乘，就等于梯度g，形式上同梯度下降的表达式是相同的。所以BFGS算法可以理解为从梯度下降逐步转换为牛顿法求函数解的一个算法。

> 虽然我们使用了BFGS算法来利用单位矩阵逐步逼近H矩阵，但是每次计算的时候都要存储D矩阵，D矩阵有多大。呢。假设我们的数据集有十万个维度(不算特别大)，那么每次迭代所要存储D矩阵的结果是74.5GB。我们无法保存如此巨大的矩阵内容，如何解决呢？使用L-BFGS算法.

我们每一次对D矩阵的迭代，都是通过迭代计算sk和yk得到的。我们的内存存不下时候只能丢掉一些存不下的数据。假设我们设置的存储向量数为100，当s和y迭代超过100时，就会扔掉第一个s和y。每多一次迭代就对应的扔掉最前边的s和y。这样虽然损失了精度，但确可以保证使用有限的内存将函数的解通过BFGS算法求得到。 所以L-BFGS算法可以理解为对BFGS算法的又一次近似或者逼近。

这里不介绍数学论证，因为网上优秀文章有很多，这里只是介绍工程实现。总结L-BFGS算法的大致步骤如下：

```
Step1:  选初始点x_0，存储最近迭代次数m;
Step2:  k=0, H_0=I, r=f(x_0);
Step3:  根据更新的参数计算梯度和损失值，如果达到阈值，则返回最优解x_{k+1}，否则转Step4;
Step4:  计算本次迭代的可行梯度下降方向 p_k=-r_k;
Step5:  计算步长alpha_k，进行一维搜索;
Step6:  更新权重x;
Step7:  只保留最近m次的向量对;
Step8:  计算并保存 s_k, y_k
```

```
Step9:  用two-loop recursion算法求r_k;
k=k+1, 转Step3。
```

0x03 优化模型 -- L-BFGS算法

回到代码， `BaseLinearModelTrainBatchOp.optimize` 函数调用的是

```
return new Lbfgs(objFunc, trainData, coefficientDim, params).optimize();
```

优化后将返回线性模型的系数。

```
/**
 * optimizer api.
 *
 * @return the coefficient of linear problem.
 */
@Override
public DataSet <Tuple2 <DenseVector, double[]>> optimize() {

   /**
    * solving problem using iteration.
    * trainData is the distributed samples.
    * initCoef is the initial model coefficient, which will be broadcast to every worker.
    * objFuncSet is the object function in dataSet format
    * .add(new PreallocateCoefficient(OptimName.currentCoef)) allocate memory for current coeff
icient
    * .add(new PreallocateCoefficient(OptimName.minCoef))     allocate memory for min loss coef
ficient
    * .add(new PreallocateLossCurve(OptimVariable.lossCurve)) allocate memory for loss values
    * .add(new PreallocateVector(OptimName.dir ...))          allocate memory for dir
    * .add(new PreallocateVector(OptimName.grad))             allocate memory for grad
    * .add(new PreallocateSkyk())                             allocate memory for sK yK
    * .add(new CalcGradient(objFunc))                         calculate local sub gradient
    * .add(new AllReduce(OptimName.gradAllReduce))            sum all sub gradient with allRedu
ce
    * .add(new CalDirection())                                get summed gradient and use it to
 calc descend dir
    * .add(new CalcLosses(objFunc, OptimMethod.GD))           calculate local losses for line s
earch
    * .add(new AllReduce(OptimName.lossAllReduce))            sum all losses with allReduce
    * .add(new UpdateModel(maxIter, epsilon ...))             update coefficient
    * .setCompareCriterionOfNode0(new IterTermination())             judge stop of iteration
    */
   DataSet <Row> model = new IterativeComQueue()
      .initWithPartitionedData(OptimVariable.trainData, trainData)
      .initWithBroadcastData(OptimVariable.model, coefVec)
      .initWithBroadcastData(OptimVariable.objFunc, objFuncSet)
      .add(new PreallocateCoefficient(OptimVariable.currentCoef))
      .add(new PreallocateCoefficient(OptimVariable.minCoef))
      .add(new PreallocateLossCurve(OptimVariable.lossCurve, maxIter))
      .add(new PreallocateVector(OptimVariable.dir, new double[] {0.0, OptimVariable.learningRa
te}))
      .add(new PreallocateVector(OptimVariable.grad))
      .add(new PreallocateSkyk(OptimVariable.numCorrections))
      .add(new CalcGradient())
      .add(new AllReduce(OptimVariable.gradAllReduce))
      .add(new CalDirection(OptimVariable.numCorrections))
      .add(new CalcLosses(OptimMethod.LBFGS, numSearchStep))
      .add(new AllReduce(OptimVariable.lossAllReduce))
      .add(new UpdateModel(params, OptimVariable.grad, OptimMethod.LBFGS, numSearchStep))
      .setCompareCriterionOfNode0(new IterTermination())
      .closeWith(new OutputModel())
      .setMaxIter(maxIter)
      .exec();

   return model.mapPartition(new ParseRowModel());
}
```

所以我们接下来的就是看Lbfgs，其重点就是参数更新的下降方向和搜索步长。

3.1 如何分布式实施

如果由于所有输入的数据都是相同维度的；算法过程中不会对输入修改，就可以将这些输入数据进行切分。这样的话，应该可以通过一次map-reduce来计算。

我们理一下L-BFGS中可以分布式并行计算的步骤：

- **计算梯度** 可以并行化 ，比如在机器1计算梯度1，在机器2计算梯度2....，然后通过一个Reduce把这些合成一个完整的梯度向量。
- **计算方向** 可以并行化，同样可以通过把数据分区，然后Map算各自分区上的值，Reduce合起来得到方向。
- **计算损失** 可以并行化，同样可以通过把数据分区，然后Map算各自分区上的值，Reduce合起来得到损失。

Alink中，使用AllReduce功能而非Flink原生Map / Reduce来完成了以上三点的并行计算和通信。

3.2 CalcGradient

线搜索只有两步，确定方向、确定步长。确定方向和模拟Hessian矩阵都需要计算梯度。目标函数的梯度向量计算中只需要进行向量间的点乘和相加，可以很容易将每个迭代过程拆分成相互独立的计算步骤，由不同的节点进行独立计算，然后归并计算结果。

Alink将样本特征向量分布到不同的计算节点，由各计算节点完成自己所负责样本的点乘与求和计算，然后将计算结果进行归并，则实现了按行并行的LR。

实际情况中也会存在针对高维特征向量进行逻辑回归的场景，仅仅按行进行并行处理，无法满足这类场景的需求，因此还需要按列将高维的特征向量拆分成若干小的向量进行求解。这个也许是Alink以后需要优化的一个点吧 。

CalcGradient是Alink迭代算子的派生类，函数总结如下：

- 获取经过处理的输入数据labledVectors，
- 从静态内存中获取 "迭代参数coef"，"优化函数objFunc" 和 "梯度"；
- 计算局部梯度 objFunc.calcGradient(labledVectors, coef, grad.f0)；这里调用到了目标函数的梯度相关API； `objFunc.calcGradient` 根据采样点计算梯度。Alink这里就是把x, y代入，求损失函数的梯度就是对 Coef求偏导数。具体我们在前文已经提到。
  - 对计算样本中的每一个样本，分别计算不同特征的计算梯度。
- 将新计算出来的梯度乘以权重之后，存入静态内存 gradAllReduce 中。
- 后续会通过聚合函数，对所有计算样本的特征的梯度进行累加，得到每一个特征的累积梯度以及损失。

```
public class CalcGradient extends ComputeFunction {

    /**
     * object function class, it supply the functions to calc local gradient (or loss).
     */
    private OptimObjFunc objFunc;

    @Override
    public void calc(ComContext context) {
        Iterable<Tuple3<Double, Double, Vector>> labledVectors = context.getObj(OptimVariable.t
rainData);

// 经过处理的输入数据
labledVectors = {ArrayList@9877}  size = 4
 0 = {Tuple3@9895} "(1.0,16.8,1.0 1.0 1.4657097546055162 1.4770978917519928)"
 1 = {Tuple3@9896} "(1.0,6.7,1.0 1.0 -0.3382407126012 73 -0.7385489458759964)"
 2 = {Tuple3@9897} "(1.0,6.9,1.0 1.0 -0.7892283294029703 -0.3692744729379982)"
 3 = {Tuple3@9898} "(1.0,8.0,1.0 1.0 -0.3382407126012 73 -0.3692744729379982)"

        // get iterative coefficient from static memory.
        Tuple2<DenseVector, Double> state = context.getObj(OptimVariable.currentCoef);
        int size = state.f0.size();

// 是Coef, 1.7976931348623157E308是默认最大值
state = {Tuple2@9878} "(0.001 0.0 0.0 0.0,1.7976931348623157E308)"
 f0 = {DenseVector@9879} "0.001 0.0 0.0 0.0"
 f1 = {Double@9889} 1.7976931348623157E308

        DenseVector coef = state.f0;
        if (objFunc == null) {
            objFunc = ((List<OptimObjFunc>)context.getObj(OptimVariable.objFunc)).get(0);
        }

// 变量如下
objFunc = {UnaryLossObjFunc@9882}
 unaryLossFunc = {SquareLossFunc@9891}
 l1 = 0.0
 l2 = 0.0
 params = {Params@9892} "Params {featureCols=["f0","f1","f2"], labelCol="label", predictionCol=
"linpred"}"

        Tuple2<DenseVector, double[]> grad = context.getObj(OptimVariable.dir);

// 变量如下
grad = {Tuple2@9952} "(0.0 0.0 0.0 0.0,[0.0, 0.1])"
 f0 = {DenseVector@9953} "0.0 0.0 0.0 0.0"
 f1 = {double[2]@9969}
coef = {DenseVector@9951} "0.001 0.0 0.0 0.0"
 data = {double[4]@9982}

        // calculate local gradient, 使用目标函数
        Double weightSum = objFunc.calcGradient(labledVectors, coef, grad.f0);

        // prepare buffer vec for allReduce. the last element of vec is the weight Sum.
        double[] buffer = context.getObj(OptimVariable.gradAllReduce);
        if (buffer == null) {
```

```
            buffer = new double[size + 1];
            context.putObj(OptimVariable.gradAllReduce, buffer);
        }

        for (int i = 0; i < size; ++i) {
            buffer[i] = grad.f0.get(i) * weightSum;
        }

      /* the last element is the weight value */
        buffer[size] = weightSum;
    }
}

// 最后结果是
buffer = {double[5]@9910}
 0 = -38.396
 1 = -38.396
 2 = -14.206109929253465
 3 = -14.364776997288134
 4 = 0.0
```

3.3 AllReduce

这里是前面提到的 "通过聚合函数，对所有计算样本的特征的梯度进行累加，得到每一个特征的累积梯度以及损失"。

具体关于AllReduce如何运作，可以参见文章 [Alink漫谈之三] AllReduce通信模型

```
.add(new AllReduce(OptimVariable.gradAllReduce))
```

3.4 CalDirection

此时得到的梯度，已经是聚合之后的，所以可以开始计算方向。

3.4.1 预先分配

OptimVariable.grad 是预先分配的。

```
public class PreallocateSkyk extends ComputeFunction {
    private int numCorrections;

    /**
     * prepare hessian matrix of lbfgs method. we allocate memory fo sK, yK at first iteration
step.
     *
     * @param context context of iteration.
     */
    @Override
    public void calc(ComContext context) {
        if (context.getStepNo() == 1) {
            Tuple2<DenseVector, double[]> grad = context.getObj(OptimVariable.grad);
            int size = grad.f0.size();
            DenseVector[] sK = new DenseVector[numCorrections];
            DenseVector[] yK = new DenseVector[numCorrections];
            for (int i = 0; i < numCorrections; ++i) {
                sK[i] = new DenseVector(size);
                yK[i] = new DenseVector(size);
            }
            context.putObj(OptimVariable.sKyK, Tuple2.of(sK, yK));
        }
    }
}
```

3.4.2 计算方向

在计算的过程中，需要不断的计算和存储历史的Hessian矩阵，在L-BFGS算法，希望只保留最近的m次迭代信息，便能够拟合Hessian矩阵。在L-BFGS算法中，不再保存完整的Hk，而是存储向量序列{sk}和{yk}，需要矩阵时Hk，使用向量序列{sk}和{yk}计算就可以得到，而向量序列{sk}和{yk}也不是所有都要保存，只要保存最新的m步向量即可。

具体原理和公式这里不再赘述，网上很多文章讲解非常好。

重点说明，dir的各个数据用途是：

```
dir = {Tuple2@9931} "(-9.599 -9.599 -3.5515274823133662 -3.5911942493220335,[4.0, 0.1])"
 f0 = {DenseVector@9954} "-9.599 -9.599 -3.5515274823133662 -3.5911942493220335" //梯度
 f1 = {double[2]@9938}
  0 = 4.0 //权重
  1 = 0.1 //学习率 learning rate, 0.1是初始化数值，后续UpdateModel时候会更新
```

代码摘要如下：

```
@Override
public void calc(ComContext context) {
```

```
   Tuple2 <DenseVector, double[]> grad = context.getObj(OptimVariable.grad);
   Tuple2 <DenseVector, double[]> dir = context.getObj(OptimVariable.dir);
   Tuple2 <DenseVector[], DenseVector[]> hessian = context.getObj(OptimVariable.sKyK);
   int size = grad.f0.size();
   // gradarr是上一阶段CalcGradient的结果
   double[] gradarr = context.getObj(OptimVariable.gradAllReduce);
// 变量为
gradarr = {double[5]@9962}
 0 = -38.396
 1 = -38.396
 2 = -14.206109929253465
 3 = -14.364776997288134
 4 = 4.0

   if (this.oldGradient == null) {
      oldGradient = new DenseVector(size);
   }
   // hessian用来当作队列，存储sK，yK，只保留最近m个
   DenseVector[] sK = hessian.f0;
   DenseVector[] yK = hessian.f1;
   for (int i = 0; i < size; ++i) {
      //gradarr[size]是权重
      grad.f0.set(i, gradarr[i] / gradarr[size]); //size = 4
   }
// 赋值梯度，这里都除以权重
grad = {Tuple2@9930} "(-9.599 -9.599 -3.5515274823133662 -3.5911942493220335,[0.0])"
 f0 = {DenseVector@9937} "-9.599 -9.599 -3.5515274823133662 -3.5911942493220335"
  data = {double[4]@9963}
   0 = -9.599
   1 = -9.599
   2 = -3.5515274823133662
   3 = -3.5911942493220335
 f1 = {double[1]@9961}
  0 = 0.0

   dir.f1[0] = gradarr[size]; //权重
   int k = context.getStepNo() - 1;

   if (k == 0) { //首次迭代
      dir.f0.setEqual(grad.f0); // 梯度赋予在这里
      oldGradient.setEqual(grad.f0);
   } else {
      yK[(k - 1) % m].setEqual(grad.f0);
      yK[(k - 1) % m].minusEqual(oldGradient);
      oldGradient.setEqual(grad.f0);
   }
   // copy g_k and store in qL

   dir.f0.setEqual(grad.f0); //拷贝梯度到这里
//
dir = {Tuple2@9931} "(-9.599 -9.599 -3.5515274823133662 -3.5911942493220335,[4.0, 0.1])"
 f0 = {DenseVector@9954} "-9.599 -9.599 -3.5515274823133662 -3.5911942493220335" //梯度
 f1 = {double[2]@9938}
  0 = 4.0 //权重
  1 = 0.1 //学习率 learning rate, 0.1是初始化数值

   // compute H^-1 * g_k
   int delta = k > m ? k - m : 0;
   int l = k <= m ? k : m; // m = 10
   if (alpha == null) {
      alpha = new double[m];
   }
   // two-loop的过程，通过拟牛顿法计算Hessian矩阵
   for (int i = l - 1; i >= 0; i--) {
      int j = (i + delta) % m;
      double dot = sK[j].dot(yK[j]);
      if (Math.abs(dot) > 0.0) {
         double rhoJ = 1.0 / dot;
         alpha[i] = rhoJ * (sK[j].dot(dir.f0)); // 计算alpha
         dir.f0.plusScaleEqual(yK[j], -alpha[i]); // 重新修正d
      }
   }
   for (int i = 0; i < l; i++) {
      int j = (i + delta) % m;
      double dot = sK[j].dot(yK[j]);
      if (Math.abs(dot) > 0.0) {
         double rhoJ = 1.0 / dot;
         double betaI = rhoJ * (yK[j].dot(dir.f0)); // 乘以rho
         dir.f0.plusScaleEqual(sK[j], (alpha[i] - betaI));// 重新修正d
      }
   }
}
```

```
//最后是存储在 OptimVariable.dir
```

3.5 CalcLosses

根据更新的 dir 和 当前系数 计算损失函数误差值，这个损失是为后续的线性搜索准备的。目的是如果损失函数误差值达到允许的范围，那么停止迭代，否则重复迭代。

CalcLosses基本逻辑如下：

- 1)得到本次步长 Double beta = dir.f1[1] / numSearchStep;
  - 后续UpdateModel 中会对下一次计算的步长（learning rate）进行更新，比如 dir.f1[1] *= 1.0 / (numSearchStep * numSearchStep); 或者 dir.f1[1] = Math.min(dir.f1[1], numSearchStep);
- 2)调用目标函数的 calcSearchValues 来计算当前系数对应的损失；
- 3)calcSearchValues 遍历输入labelVectors，对于每个 labelVector 按照线性搜索的步骤进行计算损失。vec[i] += weight * this.unaryLossFunc.loss(etaCoef - i * etaDelta, labelVector.f1); 循环内部如下：
  - 3.1)用x-vec和coef计算出来的 Y ，etaCoef = getEta(labelVector, coefVector);
  - 3.2)以x-vec和dirVec计算出来的 deltaY，etaDelta = getEta(labelVector, dirVec) * beta;
  - 3.3)按照线性搜索的步骤进行计算损失。vec[i] += weight * this.unaryLossFunc.loss(etaCoef - i * etaDelta, labelVector.f1); 联系损失函数可知，etaCoef - i * etaDelta, labelVector.f1 是 训练数据预测值 与 实际类别 的偏差；
- 4)为后续聚合 lossAllReduce 准备数据；

代码如下：

```
public class CalcLosses extends ComputeFunction {

    @Override
    public void calc(ComContext context) {
        Iterable<Tuple3<Double, Double, Vector>> labledVectors = context.getObj(OptimVariable.t
rainData);
        Tuple2<DenseVector, double[]> dir = context.getObj(OptimVariable.dir);
        Tuple2<DenseVector, Double> coef = context.getObj(OptimVariable.currentCoef);
        if (objFunc == null) {
            objFunc = ((List<OptimObjFunc>)context.getObj(OptimVariable.objFunc)).get(0);
        }
        /**
         *  calculate losses of current coefficient.
         *  if optimizer is owlqn, constraint search will used, else without constraint.
         */
        Double beta = dir.f1[1] / numSearchStep;
        double[] vec = method.equals(OptimMethod.OWLQN) ?
            objFunc.constraintCalcSearchValues(labledVectors, coef.f0, dir.f0, beta, numSearchS
tep)
            : objFunc.calcSearchValues(labledVectors, coef.f0, dir.f0, beta, numSearchStep);

// 变量为
dir = {Tuple2@9988} "(-9.599 -9.599 -3.5515274823133662 -3.5911942493220335,[4.0, 0.1])"
coef = {Tuple2@9989} "(0.001 0.0 0.0 0.0,1.7976931348623157E308)"
beta = {Double@10014} 0.025
vec = {double[5]@10015}
 0 = 0.0
 1 = 0.0
 2 = 0.0
 3 = 0.0
 4 = 0.0

        // prepare buffer vec for allReduce.
        double[] buffer = context.getObj(OptimVariable.lossAllReduce);
        if (buffer == null) {
            buffer = vec.clone();
            context.putObj(OptimVariable.lossAllReduce, buffer);
        } else {
            System.arraycopy(vec, 0, buffer, 0, vec.length);
        }
    }
}
```

其中搜索是一元目标函数提供的，其又调用了损失函数。

```
public class UnaryLossObjFunc extends OptimObjFunc {
   /**
     * Calculate loss values for line search in optimization.
     *
     * @param labelVectors train data.
     * @param coefVector   coefficient of current time.
     * @param dirVec       descend direction of optimization problem.
     * @param beta         step length of line search.
```

```
     * @param numStep      num of line search step.
     * @return double[] losses.
     */
    @Override
    public double[] calcSearchValues(Iterable<Tuple3<Double, Double, Vector>> labelVectors,
                                     DenseVector coefVector,
                                     DenseVector dirVec,
                                     double beta,
                                     int numStep) {
        double[] vec = new double[numStep + 1];

// labelVector是三元组Tuple3<weight, label, feature vector>
labelVectors = {ArrayList@10007}  size = 4
 0 = {Tuple3@10027} "(1.0,16.8,1.0 1.0 1.4657097546055162 1.4770978917519928)"
 1 = {Tuple3@10034} "(1.0,6.7,1.0 1.0 -0.338240712601273 -0.7385489458759964)"
 2 = {Tuple3@10035} "(1.0,6.9,1.0 1.0 -0.7892283294029703 -0.3692744729379982)"
 3 = {Tuple3@10036} "(1.0,8.0,1.0 1.0 -0.338240712601273 -0.3692744729379982)"
coefVector = {DenseVector@10008} "0.001 0.0 0.0 0.0"
dirVec = {DenseVector@10009} "-9.599 -9.599 -3.5515274823133662 -3.5911942493220335"
beta = 0.025
numStep = 4
vec = {double[5]@10026}
 0 = 0.0
 1 = 0.0
 2 = 0.0
 3 = 0.0
 4 = 0.0

        for (Tuple3<Double, Double, Vector> labelVector : labelVectors) {
            double weight = labelVector.f0;
            //用x-vec和coef计算出来的 Y
            double etaCoef = getEta(labelVector, coefVector);
            //以x-vec和dirVec计算出来的 deltaY
            double etaDelta = getEta(labelVector, dirVec) * beta;
weight = 1.0
etaCoef = 0.001
etaDelta = -0.7427013482280431
            for (int i = 0; i < numStep + 1; ++i) {
                //labelVector.f1就是label y
                //联系下面损失函数可知, etaCoef - i * etaDelta, labelVector.f1 是 训练数据预测值 与 实际类别 的偏差
                vec[i] += weight * this.unaryLossFunc.loss(etaCoef - i * etaDelta, labelVector.f1);
            }
        }
        return vec;
    }

    private double getEta(Tuple3<Double, Double, Vector> labelVector, DenseVector coefVector) {
      //labelVector.f2 = {DenseVector@9972} "1.0 1.0 1.4657097546055162 1.4770978917519928"
        return MatVecOp.dot(labelVector.f2, coefVector);
    }
}

vec = {double[5]@10026}
 0 = 219.33160199999998
 1 = 198.85962729259512
 2 = 179.40202828917856
 3 = 160.95880498975038
 4 = 143.52995739431051
```

回顾损失函数如下

```
/**
 * Squared loss function.
 * https://en.wikipedia.org/wiki/Loss_functions_for_classification#Square_loss
 */
public class SquareLossFunc implements UnaryLossFunc {
   public SquareLossFunc() { }

   @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;
    }
}
```

3.6 UpdateModel

本模块做两件事

- 基于dir和step length来更新coefficient，即依据方向和步长计算。
  - 如果简化理解，参数更新公式为 ： 下一刻参数 = 上一时刻参数 - 学习率 * (损失函数对这个参数的导数)。
- 判断循环的收敛。

因为变量太多，所以有时候就忘记谁是谁了，所以再次标示。

- OptimVariable.dir是CalcGradient计算出来的梯度做修正之后的结果
- OptimVariable.lossAllReduce 这个会变化，此时是上一阶段计算的损失

代码逻辑大致如下：

- 1)得出"最新损失"的最小值位置pos
- 2)得出学习率 beta = dir.f1[1] / numSearchStep;
- 3)根据"最新损失pos"和 上一个最小值 last loss value判断来进行分别处理：
  - 3.1)如果所有损失都比 last loss value 大，则
    - 3.1.1)缩减学习率 multiply 1.0 / (numSearchStep*numSearchStep)
    - 3.1.2)把 eta 设置为 0；这个就是步长了
    - 3.1.3)把当前 loss 设置为 last loss value
  - 3.2)如果losses[numSearchStep] 比 last loss value 小，则
    - 3.2.1)增大学习率 multiply numSearchStep
    - 3.2.2)设置eta是smallest value pos，eta = beta * pos; 这个eta就是步长了
    - 3.2.3)把当前 loss 设置为当前loss最小值 losses[numSearchStep]
  - 3.3)否则
    - 3.3.1)学习率不更改
    - 3.3.2)设置eta是smallest value pos，eta = beta * pos; 这个eta就是步长了
    - 3.3.3)把当前 loss 设置为当前loss最小值 losses[pos]
- 4)修正Hessian矩阵
- 5)用方向向量和步长来来更新系数向量 curCoef.f0.plusScaleEqual(dir.f0, -eta);
- 6)如果当前loss比 min loss 小，则 用 current loss 更新 min loss
  - 6.1)minCoef.f1 = curCoef.f1; 更新最小loss
  - 6.2)minCoef.f0.setEqual(curCoef.f0); 更新最小loss所对应的Coef，即线性模型最后需要的系数

在这里求步长，我没有发现 Wolf-Powell 准则的使用，Alink做了某种优化。如果有朋友能看出来Wolf-Powell如何使用，还请留言指点，谢谢。

代码如下：

```
public class UpdateModel extends ComputeFunction {
    @Override
    public void calc(ComContext context) {
        double[] losses = context.getObj(OptimVariable.lossAllReduce);
        Tuple2<DenseVector, double[]> dir = context.getObj(OptimVariable.dir);
        Tuple2<DenseVector, double[]> pseGrad = context.getObj(OptimVariable.pseGrad);
        Tuple2<DenseVector, Double> curCoef = context.getObj(OptimVariable.currentCoef);
        Tuple2<DenseVector, Double> minCoef = context.getObj(OptimVariable.minCoef);

        double lossChangeRatio = 1.0;
        double[] lossCurve = context.getObj(OptimVariable.lossCurve);
        for (int j = 0; j < losses.length; ++j) {
            losses[j] /= dir.f1[0]; //dir.f1[0]是权重
        }
        int pos = -1;
        //get the min value of losses, and remember the position.
        for (int j = 0; j < losses.length; ++j) {
            if (losses[j] < losses[0]) {
                losses[0] = losses[j];
                pos = j;
            }
        }

        // adaptive learningRate strategy
        double beta = dir.f1[1] / numSearchStep;
        double eta;

// 变量如下
losses = {double[5]@10001}
 0 = 35.88248934857763
 1 = 49.71490682314878
 2 = 44.85050707229464
```

```
 3 = 40.239701247437594
 4 = 35.88248934857763
dir = {Tuple2@10002} "(-9.599 -9.599 -3.5515274823133662 -3.5911942493220335,[4.0, 0.1])"
pseGrad = null
curCoef = {Tuple2@10003} "(0.001 0.0 0.0 0.0,1.7976931348623157E308)"
minCoef = {Tuple2@10004} "(0.001 0.0 0.0 0.0,1.7976931348623157E308)"
lossChangeRatio = 1.0
lossCurve = {double[100]@10005}
pos = 4
beta = 0.025
curCoef.f1 = {Double@10006} 1.7976931348623157E308

        if (pos == -1) {
            /**
             * if all losses larger than last loss value. we'll do the below things:
             * 1. reduce learning rate by multiply 1.0 / (numSearchStep*numSearchStep).
             * 2. set eta with zero.
             * 3. set current loss equals last loss value.
             */
            eta = 0;
            dir.f1[1] *= 1.0 / (numSearchStep * numSearchStep);  // 学习率
            curCoef.f1 = losses[0]; // 最小loss
        } else if (pos == numSearchStep) {
            /**
             * if losses[numSearchStep] smaller than last loss value. we'll do the below things
:
             * 1. enlarge learning rate by multiply numSearchStep.
             * 2. set eta with the smallest value pos.
             * 3. set current loss equals smallest loss value.
             */
            eta = beta * pos;
            dir.f1[1] *= numSearchStep;
            dir.f1[1] = Math.min(dir.f1[1], numSearchStep); // 学习率
            lossChangeRatio = Math.abs((curCoef.f1 - losses[pos]) / curCoef.f1);
            curCoef.f1 = losses[numSearchStep]; // 最小loss

// 当前数值
numSearchStep = 4 是NUM_SEARCH_STEP缺省值，是线性搜索的参数，Line search parameter, which define the
 search value num of one step.
lossChangeRatio = 1.0
pos = 4
eta = 0.1
curCoef.f1 = {Double@10049} 35.88248934857763
dir.f1[1] = 0.4

        } else {
            /**
             * else :
             * 1. learning rate not changed.
             * 2. set eta with the smallest value pos.
             * 3. set current loss equals smallest loss value.
             */
            eta = beta * pos;
            lossChangeRatio = Math.abs((curCoef.f1 - losses[pos]) / curCoef.f1);
            curCoef.f1 = losses[pos]; // 最小loss
        }

                /* update loss value in loss curve at this step */
        lossCurve[context.getStepNo() - 1] = curCoef.f1;

lossCurve = {double[100]@9998}
 0 = 35.88248934857763
 1 = Infinity

        if (method.equals(OptimMethod.OWLQN)) {
           .....
        } else if (method.equals(OptimMethod.LBFGS)) {
            Tuple2<DenseVector[], DenseVector[]> sKyK = context.getObj(OptimVariable.sKyK);
            int size = dir.f0.size();
            int k = context.getStepNo() - 1;
            DenseVector[] sK = sKyK.f0;
            for (int s = 0; s < size; ++s) {
                // 修正矩阵
                sK[k % OptimVariable.numCorrections].set(s, dir.f0.get(s) * (-eta));
            }
            curCoef.f0.plusScaleEqual(dir.f0, -eta); // 这里是用方向向量和步长来更新系数向量

sKyK = {Tuple2@10043} "([0.9599000000000001 0.9599000000000001 0.35515274823133663 0.3591194249
322034, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.
0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0],[0.0 0.0 0.0 0.0, 0.0 0.0 0.0
 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0
0.0 0.0 0.0, 0.0 0.0 0.0 0.0, 0.0 0.0 0.0 0.0])"
```

```
 f0 = {DenseVector[10]@10044}
  0 = {DenseVector@10074} "0.9599000000000001 0.9599000000000001 0.35515274823133663 0.3591194249322034"

        }

        /**
         * if current loss is smaller than min loss, then update the min loss and min coefficient by current.
         */
        if (curCoef.f1 < minCoef.f1) {
            minCoef.f1 = curCoef.f1; // 最小loss
            minCoef.f0.setEqual(curCoef.f0); // 最小loss所对应的Coef，即线性模型最后需要的系数
        }

curCoef = {Tuple2@9996} "(0.9609000000000001 0.9599000000000001 0.35515274823133663 0.3591194249322034,35.88248934857763)"
 f0 = {DenseVector@10059} "0.9609000000000001 0.9599000000000001 0.35515274823133663 0.3591194249322034"
 f1 = {Double@10048} 35.88248934857763
minCoef = {Tuple2@9997} "(0.9609000000000001 0.9599000000000001 0.35515274823133663 0.3591194249322034,35.88248934857763)"
 f0 = {DenseVector@10059} "0.9609000000000001 0.9599000000000001 0.35515274823133663 0.3591194249322034"
 f1 = {Double@10048} 35.88248934857763

        // judge the convergence of iteration.
        filter(dir, curCoef, minCoef, context, lossChangeRatio);
    }
}
```

filter 判断是否收敛，里面的打印log很清晰的说明了函数逻辑。

```
/**
 * judge the convergence of iteration.
 */
public void filter(Tuple2<DenseVector, double[]> grad,
                   Tuple2<DenseVector, Double> c,
                   Tuple2<DenseVector, Double> m,
                   ComContext context,
                   double lossChangeRatio) {
    double epsilon = params.get(HasEpsilonDv0000001.EPSILON);
    int maxIter = params.get(HasMaxIterDefaultAs100.MAX_ITER);
    double gradNorm = ((Tuple2<DenseVector, double[]>)context.getObj(gradName)).f0.normL2();
    if (c.f1 < epsilon || gradNorm < epsilon) {
        printLog(" method converged at step : ", c.f1, m.f1, grad.f1[1], gradNorm, context, lossChangeRatio);
        grad.f1[0] = -1.0;
    } else if (context.getStepNo() > maxIter - 1) {
        printLog(" method stop at max step : ", c.f1, m.f1, grad.f1[1], gradNorm, context, lossChangeRatio);
        grad.f1[0] = -1.0;
    } else if (grad.f1[1] < EPS) {
        printLog(" learning rate is too small, method stops at step : ", c.f1, m.f1, grad.f1[1], gradNorm,
            context, lossChangeRatio);
        grad.f1[0] = -1.0;
    } else if (lossChangeRatio < epsilon && gradNorm < Math.sqrt(epsilon)) {
        printLog(" loss change ratio is too small, method stops at step : ", c.f1, m.f1, grad.f1[1], gradNorm,
            context, lossChangeRatio);
        grad.f1[0] = -1.0;
    } else {
        printLog(" method continue at step : ", c.f1, m.f1, grad.f1[1], gradNorm, context, lossChangeRatio);
    }
}
```

3.7 OutputModel

`.closeWith(new OutputModel())` 这部分是每次迭代结束，临时输出模型，把数据转换成Flink通用的Row类型，Transfer the state to model rows。

```
public class OutputModel extends CompleteResultFunction {

   @Override
   public List <Row> calc(ComContext context) {
      // get the coefficient of min loss.
      Tuple2 <DenseVector, double[]> minCoef = context.getObj(OptimVariable.minCoef);
      double[] lossCurve = context.getObj(OptimVariable.lossCurve);

      int effectiveSize = lossCurve.length;
```

```
      for (int i = 0; i < lossCurve.length; ++i) {
         if (Double.isInfinite(lossCurve[i])) {
            effectiveSize = i;
            break;
         }
      }

      double[] effectiveCurve = new double[effectiveSize];
      System.arraycopy(lossCurve, 0, effectiveCurve, 0, effectiveSize);

      Params params = new Params();
      params.set(ModelParamName.COEF, minCoef.f0);// 重点在这里，minCoef是我们真正需要的
      params.set(ModelParamName.LOSS_CURVE, effectiveCurve);
      List <Row> model = new ArrayList <>(1);
      model.add(Row.of(params.toJson()));
      return model;
   }
}
```

0x04 准备模型元数据

这里设置了并行度为1。

```
// Prepare the meta info of linear model.
DataSet<Params> meta = labelInfo.f0
    .mapPartition(new CreateMeta(modelName, linearModelType, isRegProc, params))
    .setParallelism(1);
```

具体代码

```
public static class CreateMeta implements MapPartitionFunction<Object, Params> {
        @Override
        public void mapPartition(Iterable<Object> rows, Collector<Params> metas) throws Excepti
on {
            Object[] labels = null;
            if (!this.isRegProc) {
                labels = orderLabels(rows);
            }

            Params meta = new Params();
            meta.set(ModelParamName.MODEL_NAME, this.modelName);
            meta.set(ModelParamName.LINEAR_MODEL_TYPE, this.modelType);
            meta.set(ModelParamName.LABEL_VALUES, labels);
            meta.set(ModelParamName.HAS_INTERCEPT_ITEM, this.hasInterceptItem);
            meta.set(ModelParamName.VECTOR_COL_NAME, vectorColName);
            meta.set(LinearTrainParams.LABEL_COL, labelName);
            metas.collect(meta);
        }
}

// 变量为
meta = {Params@9667} "Params {hasInterceptItem=true, vectorColName=null, modelName="Linear Regr
ession", labelValues=null, labelCol="label", linearModelType="LinearReg"}"
```

0x05 建立模型

当迭代循环结束之后，Alink就根据Coef数据来建立模型。

```
/**
 * build the linear model rows, the format to be output.
 */
public static class BuildModelFromCoefs extends AbstractRichFunction implements
        @Override
        public void mapPartition(Iterable<Tuple2<DenseVector, double[]>> iterable,
                                 Collector<Row> collector) throws Exception {
            for (Tuple2<DenseVector, double[]> coefVector : iterable) {
                LinearModelData modelData = buildLinearModelData(meta,
                    featureNames,
                    labelType,
                    meanVar,
                    hasIntercept,
                    standardization,
                    coefVector);

                new LinearModelDataConverter(this.labelType).save(modelData, collector);
            }
        }
}
```

得到模型数据为，里面coef就是 f(x)=w^Tx+b 中的 w, b。是最终用来计算的。

```
modelData = {LinearModelData@10584}
```

```
 featureNames = {String[3]@9787}
 featureTypes = null
 vectorColName = null
 coefVector = {DenseVector@10485} "-3.938937407856857 4.799499941426075 0.8929571907809862 1.07
8169576770847"
 coefVectors = null
 vectorSize = 3
 modelName = "Linear Regression"
 labelName = null
 labelValues = null
 linearModelType = {LinearModelType@4674} "LinearReg"
 hasInterceptItem = true
 lossCurve = {double[12]@10593}
  0 = 35.88248934857763
  1 = 12.807366842002144
  2 = 0.5228366663917704
  3 = 0.031112070740366038
  4 = 0.01098914933042993
  5 = 0.009765757443537283
  6 = 0.008750523231785415
  7 = 0.004210085397869248
  8 = 0.0039042232755530704
  9 = 0.0038821509860327537
  10 = 0.003882042680010676
  11 = 0.0038820422536391033
 labelType = {FractionalTypeInfo@9790} "Double"
```

0x06 使用模型预测

预测时候，使用的是LinearModelMapper，其内部部分变量打印出来如下，能够看出来模型数据。

```
this = {LinearModelMapper@10704}
 vectorColIndex = -1
model = {LinearModelData@10597}
 featureNames = {String[3]@10632}
 featureTypes = null
 vectorColName = null
 coefVector = {DenseVector@10612} "-3.938937407856857 4.799499941426075 0.8929571907809862 1.07
8169576770847"
 coefVectors = null
 vectorSize = 0
 modelName = "Linear Regression"
 labelName = null
 labelValues = {Object[0]@10613}
 linearModelType = {LinearModelType@10611} "LinearReg"
 hasInterceptItem = true
 lossCurve = null
 labelType = null
```

具体预测是在 `LinearModelMapper.predict` 中完成，具体如下：

- 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),` ，
- 通过 `FeatureLabelUtil.getFeatureVector` 处理之后，
- 得到的四元组是 `"1.0 1.0 7.0 9.0"` ，其中第一个 1.0 是通过 `aVector.set(0, 1.0);` 专门设定的固定值。比如模型是 f(x) = ax + b，这个固定值 1.0 就是 b 的初始化值，随着优化过程会得到 b。所以这里还是需要有一个 1.0 来进行预测。
- 模型系数是： `"-3.938937407856857 4.799499941426075 0.8929571907809862 1.078169576770847"` 。
- 四元组 和 模型系数 点积的结果就是 `dotValue = 16.814789059973744` 。

这样就能看出来模型系数如何使用的了。

```
public Object predict(Vector vector) throws Exception {
   double dotValue = MatVecOp.dot(vector, model.coefVector);

   switch (model.linearModelType) {
      case LR:
      case SVM:
      case Perceptron:
         return dotValue >= 0 ? model.labelValues[0] : model.labelValues[1];
      case LinearReg:
      case SVR:
         return dotValue;
   }
}

vector = {DenseVector@10610} "1.0 1.0 7.0 9.0"
 data = {double[4]@10619}
  0 = 1.0
  1 = 1.0
  2 = 7.0
```

```
  3 = 9.0
model.coefVector = {DenseVector@10612} "-3.938937407856857 4.799499941426075 0.8929571907809862
 1.078169576770847"
 data = {double[4]@10620}
  0 = -3.938937407856857
  1 = 4.799499941426075
  2 = 0.8929571907809862
  3 = 1.078169576770847
```

0x07 本系列其他文章

Alink漫谈(一)：从KMeans算法实现不同看Alink设计思想

Alink漫谈(二)：从源码看机器学习平台Alink设计和架构

[Alink漫谈之三] AllReduce通信模型

Alink漫谈(四)：模型的来龙去脉

Alink漫谈(五)：迭代计算和Superstep

Alink漫谈(六)：TF-IDF算法的实现

Alink漫谈(七)：如何划分训练数据集和测试数据集

Alink漫谈(八)：二分类评估 AUC、K-S、PRC、Precision、Recall、LiftChart 如何实现

Alink漫谈(九)： 特征工程 之 特征哈希/标准化缩放

Alink漫谈(十)： 线性回归实现 之 数据预处理

0xFF 参考

http://www.mamicode.com/info-detail-2508527.html)

机器学习算法实现解析——liblbfgs之L-BFGS算法

深入机器学习系列之BFGS & L-BFGS

拟牛顿法公式推导以及python代码实现（一）

浅显易懂——泰勒展开式

泰勒展开 — Taylor Expansion

从牛顿插值法到泰勒公式

怎样更好地理解并记忆泰勒展开式?

优化算法——牛顿法(Newton Method)

优化算法——拟牛顿法之L-BFGS算法

一文读懂L-BFGS算法

《分布式机器学习算法、理论与实践 刘铁岩》

LogisticRegressionWithLBFGS--逻辑回归

逻辑回归优化方法-L-BFGS

机器学习与运筹优化（三）从牛顿法到L-BFGS

机器学习中牛顿法凸优化的通俗解释

深入机器学习系列17-BFGS & L-BFGS

优化方法基础系列-精确的一维搜索技术

优化方法基础系列-非精确的一维搜索技术

[原创]用"人话"解释不精确线搜索中的Armijo-Goldstein准则及Wolfe-Powell准则

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

一维搜索算法介绍及其实现

数值优化|笔记整理（1）——引入，线搜索：步长选取条件

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

线搜索（一）：步长的选取

最优化问题——一维搜索(二)

CRF L-BFGS Line Search原理及代码分析

步长与学习率

机器学习优化算法L-BFGS及其分布式实现

L-BFGS算法详解（逻辑回归的默认优化算法）

线性回归的原理及实践（牛顿法）

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

L-BFGS算法

并行逻辑回归