Logistic Regression In Numerical Optimization

内部版本，未写完，不能群发

作者：马晨

Chapter 1 简介 Logistic Regression introduction

如果将 $p(x)$作为 x 的线性函数来计算会有以下问题：

1. $p(x)$的范围为[0, 1]，而线性函数的范围为[-∞, +∞]
2. 在很多情况下我们会遇到"diminishing return"，而线性模型不满足该特性
   Def. diminishing return: changing $p(x)$ by the same amount requires a bigger change in x when $p(x)$ is already large (or small) than when $p(x)$ is close to 1/2.

如果将 log$p(x)$作为 x 的线性函数，则会有以下问题：
因为 $p(x) \in [0, 1]$，所以其对数函数 log$p(x) \in [-\infty, 0]$，而线性函数的范围为[-∞, +∞]

所以如果需要函数的值域属于[-∞, +∞]，则可以采用 logistic (or logit) transformation，即：

$$\text{logistic transformation: } \log\frac{p(x)}{1-p(x)}$$

即为 **logistic regression** 的 sigmoid 函数:

$$\log\frac{p(x)}{1-p(x)} = \beta_0 + \boldsymbol{x}\cdot\boldsymbol{\beta} \rightarrow p(x) = \frac{e^{\beta_0+\boldsymbol{x}\cdot\boldsymbol{\beta}}}{1+e^{\beta_0+\boldsymbol{x}\cdot\boldsymbol{\beta}}} = \frac{1}{1+e^{-(\beta_0+\boldsymbol{x}\cdot\boldsymbol{\beta})}}$$

当 $p(x) \geq 0.5$，即$\beta_0 + \boldsymbol{x}\cdot\boldsymbol{\beta}$为非负时，Y = 1；当 $p(x) < 0.5$，即$\beta_0 + \boldsymbol{x}\cdot\boldsymbol{\beta}$为负时，Y = 0。

由于 sigmoid 函数可以成功地将任意实值映射到[0,1]，所以可以将其看作概率，这一点有时非常有用。

此时可以尝试使用 Maximum Likelihood Estimation(MLE)来做参数估计，LR 的 likelihood function 如下所示：

$$L(\beta_0,\beta) = \prod_{i=1}^{n} p(x_i)^{y_i}(1-p(x_i)^{1-y_i}$$

将上式转换为 log-likelihood function 如下：

$$\begin{aligned}\ell(\beta_0,\beta) &= \sum_{i=1} y_i \log p(x_i) + (1-y_i)\log 1-p(x_i) \\ &= \sum_{i=1}^{n} \log 1-p(x_i) + \sum_{i=1}^{n} y_i \log \frac{p(x_i)}{1-p(x_i)} \\ &= \sum_{i=1}^{n} \log 1-p(x_i) + \sum_{i=1}^{n} y_i(\beta_0 + x_i \cdot \beta) \\ &= \sum_{i=1}^{n} -\log 1+e^{\beta_0+x_i\cdot\beta} + \sum_{i=1}^{n} y_i(\beta_0 + x_i \cdot \beta)\end{aligned}$$

要求得第 j 个 feature 的参数值 β_j，计算上式对于 β_j 的偏导等于零，即：

$$\begin{aligned}\frac{\partial \ell}{\partial \beta_j} &= -\sum_{i=1}^{n} \frac{1}{1+e^{\beta_0+x_i\cdot\beta}} e^{\beta_0+x_i\cdot\beta} x_{ij} + \sum_{i=1}^{n} y_i x_{ij} \\ &= \sum_{i=1}^{n} (y_i - p(x_i;\beta_0,\beta))\, x_{ij}\end{aligned}$$

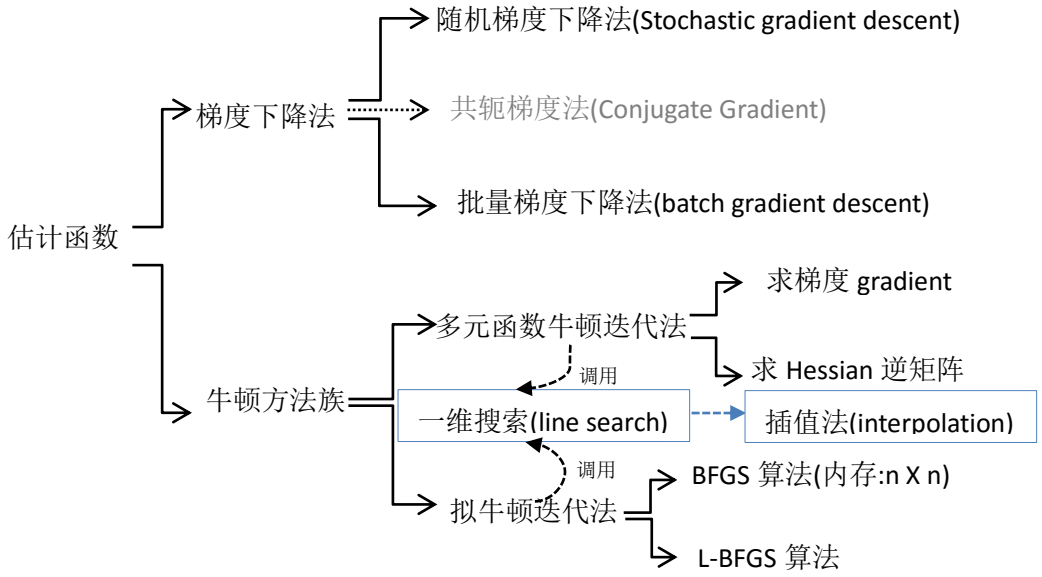
如果上式等于零，则是一个**超越方程**，并不能得到它的解。所以需要通过其他方式来近似求各个特征的参数值。

所以为了求解该 LR 的 log-likehood function 的 MLE（极大似然估计），可以采用的参数估计方法(Parameter Estimate)有**梯度下降法**和**牛顿/拟牛顿迭代法**。

下文中不论是梯度下降法还是牛顿或拟牛顿迭代法，实质都是解决一个问题：**如何（更快地）求无约束情况下的函数的极小值。**

在这里的 MLE 中，可以在 log-likehood function 前加上负号（也可以使用损失函数 cost function），让求极大值问题转变为极小值。

所以整体章节安排按照下图所示，其中灰色字体的共轭梯度法不在这一版本的作品中。在讲牛顿方法之前，我们单独安排一个章节先介绍里面用到的一维搜索(line search)技术。



Chapter 2 梯度下降法(gradient descent)

2.1 Derivative LR's log-likehood function

设有假设 $h_\theta(x)$如下：

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

其中

$$g(z) = \frac{1}{1+e^{-z}} : sigmoid\ \ function$$

Sigmoid function 有如下特性：

$$g'(z) = \frac{d}{dz}\frac{1}{1+e^{-z}} = \frac{1}{(1+e^{-z})^2}(e^{-z})$$

$$= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{1+e^{-z}}\right) = g(z)(1-g(z))$$

我们假设：

$$P(y=1|x;\theta) = h_\theta(x)$$
$$P(y=0|x;\theta) = 1 - h_\theta(x)$$

即：

$$P(y|x;\theta) = \left(h_\theta(x)\right)^y (1-h_\theta(x))^{1-y}$$

明确几个公式中的符号：

m：训练样本行数，共 m 个训练样本
n：待估计的参数个数，也即每行 x 的列数

$\vec{\theta}$ ：待估计的参数，共 n 个参数

$\vec{\mathrm{x}}$ ：每行的数据值，如果第 i 行数据中的 j 个 x 值为 $x_j^{(i)}$

$\vec{\theta}^{\mathrm{T}} \bullet \vec{\mathrm{x}} = \theta_1 \cdot \mathrm{x}_1 + \theta_2 \cdot \mathrm{x}_2 + \theta_3 \cdot \mathrm{x}_3 + \ldots + \theta_n \cdot \mathrm{x}_n$

假设有 m 个训练样本，且互相之间独立同分布（IID），则参数的似然函数为：

$$L(\theta) = \prod_{i=1}^{m} P\left(y^{(i)}|x^{(i)};\theta\right) = \prod_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right)\right)^{y^{(i)}} \left(1 - h_\theta\left(x^{(i)}\right)\right)^{1-y^{(i)}}$$

上式的对数似然函数为：

$$l(\theta) = logL(\theta) = \sum_{i=1}^{m} (y^{(i)} log h_\theta\left(x^{(i)}\right) + (1-y^{(i)}) \log(1 - h_\theta\left(x^{(i)}\right))$$

现在的目标是要最大化上式的对数似然函数。此时使用梯度上升[1]，即$\theta := \theta + \alpha\nabla_\theta l(\theta)$。对一个参数 θ_j 有：

$$\begin{aligned}\frac{\partial}{\partial\theta_j}\ell(\theta) &= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right)\frac{\partial}{\partial\theta_j}g(\theta^T x)\\ &= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right)g(\theta^T x)(1-g(\theta^T x)\frac{\partial}{\partial\theta_j}\theta^T x\\ &= \left(y(1-g(\theta^T x)) - (1-y)g(\theta^T x)\right)x_j\\ &= \left(y-h_\theta(x)\right)x_j\end{aligned}$$

上面的推导中使用了 sigmoid function 的特性：$g'(z) = g(z)(1-g(z))$和随机梯度上升。最终我们得到的随机梯度上升的参数估计规则为：

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$$

相比于梯度上升（batch），一般说来牛顿方法能够更快地收敛，即牛顿方法的循环次数要少于梯度上升。但是，牛顿方法每次循环的计算量要高于梯度上升，这是因为在牛顿方法中每次循环都需要建立和计算 Hessian 矩阵及其逆矩阵。

2.2 关于梯度下降/上升中的 batch and stochastic

以梯度下降为例（梯度上升基本一致），其公式如下：

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta)$$

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

其中 $J(\theta)$为损失函数(least-squares cost function)。假设训练集中只有一个样本，则有：

$$\begin{aligned}\frac{\partial}{\partial\theta_j}J(\theta) &= \frac{\partial}{\partial\theta_j}\frac{1}{2}(h_\theta(x)-y)^2\\ &= 2\cdot\frac{1}{2}(h_\theta(x)-y)\cdot\frac{\partial}{\partial\theta_j}(h_\theta(x)-y)\\ &= (h_\theta(x)-y)\cdot\frac{\partial}{\partial\theta_j}\left(\sum_{i=0}^{n}\theta_i x_i - y\right)\\ &= (h_\theta(x)-y)\,x_j\end{aligned}$$

即：

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$$

当训练集的样本有 m 个时(m > 1)，此时实现梯度下降有两种方式：

Batch gradient descent/批量梯度下降

对于每个参数 θ_j，使用全量样本对其做梯度下降，直至收敛，即如下所示：

[1] 梯度本意是一个向量（矢量），当某一函数在某点处沿着该方向的方向导数取得该点处的最大值，即函数在该点处沿方向变化最快，变化率最大（为该梯度的模）。

Repeat until convergence {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

Stochastic gradient descent/随机梯度下降

对于每个样本，对所有的参数 θ_j 做一次梯度下降计算，直至收敛，即如下所示：

Loop {

for i=1 to m, {

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

}

相比于批量梯度下降，随机梯度下降能够更快地逼近真实参数结果，所以，当训练样本数比较大时，经常使用随机梯度下降来计算。

2.3 实现

我以文本分类作为例子，来展示 python 语言实现一个梯度下降法文本 LR 分类器的例子，在这个例子中，文章的特征以词 wordid、以及每个词在该篇文章上的 TF-IDF[2]来表征该词的特征。例子代码如下：

```
from collections import defaultdict
import math
import heapq
import random

def inner_product(x,y):
    sum = 0.0
    for _x, _score in x.iteritems():
        sum += _score * y[_x]
    return sum

def sigmoid(x,y,inner=None):
    if inner is None:
        inner = inner_product(x,y)
    #don't use bias theta 0, directly use inner product
    result = 1.0 / (1+ math.e ** (-(inner)))
    return result

#主要函数：theta 变量为要训练的权重
def lr_stochastic_gradient(tfidfmatrix, theta, predict_tfidf):
```

[2] TF-IDF 提取文本特征的代码已经略去，如果想了解：http://www.ruanyifeng.com/blog/2013/03/tf-idf.html

```
    alpha = 0.01
    for iter in xrange(100):  #100 次迭代
        print "iter in %s"%(iter)
        for tfidf,y in tfidfmatrix:  #y 值=0 或 1，tfidf 的 key 是 wordid，value 是该 wordid 的 TF-IDF 值
            #print "train y:%s"%(y)
            inner = inner_product(tfidf, theta)
            last_product_item = 0
            for wordid, score in tfidf.iteritems():
                #last_product_item = theta[wordid] * score
                theta[wordid] = theta[wordid] + alpha * (y - sigmoid(tfidf, theta, inner)) * score

#训练完成，使用 theta 权重来预测分类
def lr_predict(predict_tfidf, theta):
    # use train set to test predict
    print "train over, begin predict test file"
    predict_right = 0
    recall_denominator = 0
    recall_numerator = 0
    for tfidf,orig_y in predict_tfidf:
        theta_score = sigmoid(tfidf, theta)
        new_y = 0
        if theta_score >= 0.5:
            new_y = 1
        if new_y == orig_y:
            predict_right += 1
        if orig_y == 0:
            recall_denominator += 1
            if new_y == 0:
                recall_numerator += 1
    print "precision:%s recall:%s"%  ￥
        (float(predict_right) / len(predict_tfidf),
         recall_numerator / float(recall_denominator))
```

2.4 总结

缺点：1.迭代速度慢，每次走固定 step size 的速率

2.在靠近极小值时走“之字形”，也就是说越接近目标，步长越小，前进越慢。

注意到迭代公式里有个 α 的步长，α 如果设置过小，则迭代速度太慢；如果 α 设置过大，则可能步子迈得太大，造成不能到达极小值点。下一章我们讨论的一维搜索（line search）技术解决这个问题的。

Chapter 3 一维搜索(Line Search)

Line search 解决这样一个看似简单的问题[3]，就是自动确定学习步长 α，通常，方向角度由梯度下降法或牛顿（拟牛顿）迭代法得出，而每一次前进的**合适的**学习步长就要靠 line search 技术了。

我们在下述推导前，先明确几个公式中用到的符号：

P_k：函数的前进方向（牛顿迭代法中为$-H^{-1}(\theta^{(n)})\cdot \nabla f(\theta^{(n)})$，在迭代法中确定，一维搜索中为常量）
$f(x_k+\alpha_k P_k)$：即 LR 中的-log-likehood function，整本书的想要找到极值点的函数。
$\Phi(\alpha)$: 将原本的 $f(x_k+\alpha_k P_k)$改为以 α 作为自变量的函数，而因变量值仍旧不变。

我们设计了一个函数[4] ：$\Phi(\alpha) = f(x_k+\alpha_k P_k)$

那么本章的目标就是一个简单的问题:**如何选择 α^*使得 $\Phi(\alpha)$达到极小值。**

这里的难点在于我们并不知道 $\Phi(\alpha)$的函数的表达形式[5]，因此无法使用求导使之为 0 的方法来求那个 α^*，因此引出了一维搜索技术的 wolfe condition。

3.1 wolfe condition

为了保证 Φ(α)达到的极小值，我们的函数 f 需要满足一些条件，只要找到满足这些条件的 α 区间，就可以在这些区间内找到 α^*。

3.1.1 Armiji condition

第一个介绍的条件是 Armiji condition，也称之为 Sufficient Decrease Condition（充分下降条件）。该条件如下：

$$f(x_k+\alpha P_k) \le f(x_k)+c_1\alpha\cdot\nabla f_k^T\cdot P_k$$

$$其中 c_1\in(0,1)$$

如果你还记得中学的直线的斜截式方程 f(x)=k·x+b，其中 k 是斜率，b 是截距。

则不等式右边的式子也可以等价于直线 $l(\alpha)=\underbrace{c_1\cdot\overbrace{\nabla f_k^T\cdot P_k}^{<0}}_{k,k<0}\cdot\alpha+\underbrace{f(x_k)}_{b}$[6]

如下图所示：

[3] 其实在代码实现中就是一个返回 α 的 line search 函数

[4] 无论函数 f(x)是几维(几元)函数，Φ(α)都只有一个 α 的一维(一元)函数

[5] 但是我们可以代入 α 获得 Φ(α)的值，关于 LR 中的 Φ(α)在 3.3 节中有介绍

[6] α 每前进一步，l(α)都减小$|c_1\cdot\nabla f_k^T\cdot P_k|$

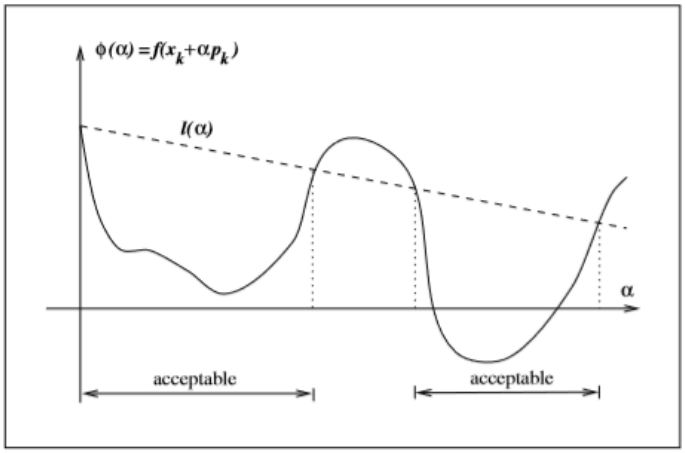


图 3.1 Armiji condition

注意此图的横坐标是 α，以 α 作为自变量，**而 x_k，P_k，∇f_k 都是常量，**自然该曲线 Φ(α) 与 f(x)的图像有所不同，这个图的意思已经明了：α^*只能落在该直线与 Φ(α)相交包围**以下**的部分。

3.1.2 Curvature condition

Curvature condition 又被称为斜率条件，这个条件如下：

$$\nabla f(x_k+\alpha_k P_k)^T \cdot P_k \geq c_2 \cdot \nabla f_k^T \cdot P_k$$

$$其中 c_2 \in (c_1,1)$$

注意不等式的左边=Φ′(α)，因为 $\Phi'(\alpha)=\dfrac{d\Phi(\alpha)}{d\alpha}=\nabla f(x_k+\alpha_k P_k)^T \cdot P_k$。

不等式的右边=c_2·Φ′(0)，因为 $\Phi'(0)=\left.\dfrac{d\Phi(\alpha)}{d\alpha}\right|_{\alpha=0}=\nabla f(x_k)^T \cdot P_k$

即 $\Phi'(\alpha) \geq c_2 \cdot \Phi'(0)$

如下图：

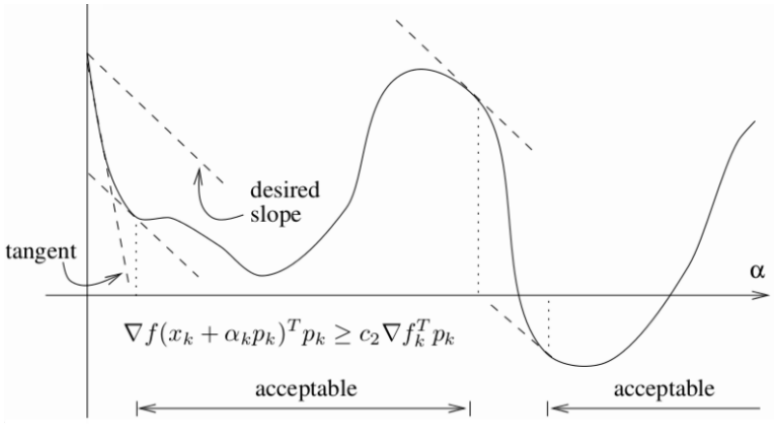


图 3.2 Curvature condition

此条件的含义从这张图可以明了，斜率条件的意思是说：由于在极小值点处$\Phi'(\alpha^*)=0$，因此接近位置时的斜率（即导数）应该比初始位置时**更平缓，而初始位置时更陡或更倾斜。由于斜率小于 0，所以应该更大（$\phi'(\alpha)$ 甚至可能大于 0 [7]），于是引出了这个不等式。**

这里 c_2 的作用就是让 Φ′(0)（其<0）更平缓一些再作为不等式的条件。

3.1.3 two condition combine

上述两个条件联立，就是 wolfe condition 了。

$$f(x_k+\alpha P_k)\le f(x_k)+c_1\alpha\cdot\nabla f_k^T\cdot P_k$$

$$\nabla f(x_k+\alpha_k P_k)^T\cdot P_k\ge c_2\cdot\nabla f_k^T\cdot P_k$$

$$0<c_1<c_2<1$$

第一个条件主要限制步长不能太大，第二条主要限制步长不能太短。
如下图所示：

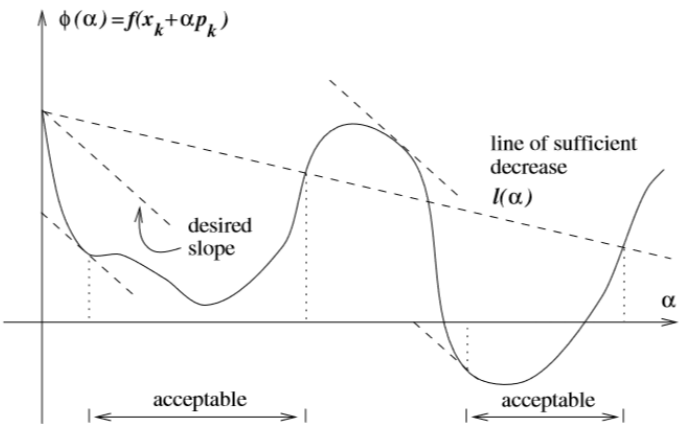


图 3.3 wolfe condition[8]

这里我们必须回答几个问题，以便深入理解：
Q&A：

1. 为何有$0<c_1<c_2<1$这个条件？

   因为 Armiji condition 中的$l(\alpha)=\underbrace{c_1\cdot\overbrace{\nabla f_k^T\cdot P_k}^{<0}}_{k,k<0}\cdot\alpha+\underbrace{f(x_k)}_{b}$的斜率为$c_1\cdot\nabla f_k^T\cdot P_k$，而 Curvature condition 中的要满足的曲线斜率的必须比 $c_2\cdot\Phi'(\alpha)=c_2\cdot\nabla f_k^T\cdot P_k$ 更平缓，注意到两者的导数皆包含$\nabla f_k^T\cdot P_k$，这也就把 c_1 和 c_2 联系起来了。也就是说只有

[7] 这就引出了 stong wolfe condition
[8] Wolfe condition 图片均来自《Numerical Optimization》（Jorge Nocedal & Stephen J. Wright）

$c_1 < c_2$ 时，才有“**$l(\alpha)$ 直线平缓一些，曲线斜率条件往下更陡些，直线 $l(\alpha)$ 和曲线 Φ(α)才能有围成交集，进而找到满足 α^* 的区间**”，否则 $c_1 > c_2$，就找不到围出来的交集了。

从下面这个图可以看的更明显：

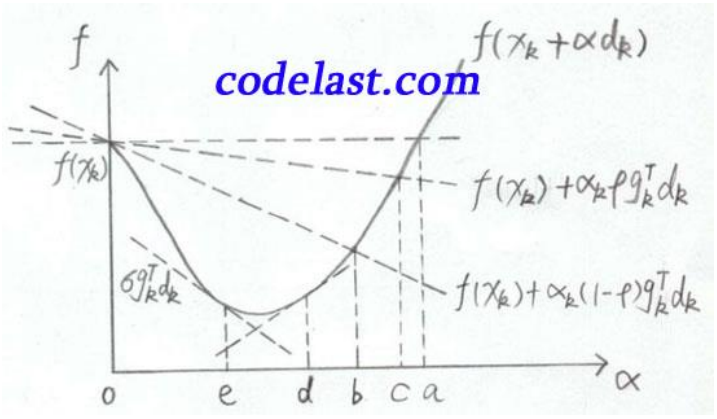


图 3.4 why $0 < c_1 < c_2 < 1$

2．什么是 strong wolfe condition

将第二个条件改为绝对值（并且改变不等符号方向）：

$$f(x_k + \alpha P_k) \le f(x_k) + c_1 \alpha \cdot \nabla f_k^T \cdot P_k$$

$$|\nabla f(x_k + \alpha_k P_k)^T \cdot P_k| \le c_2 \cdot |\nabla f_k^T \cdot P_k|$$

$$0 < c_1 < c_2 < 1$$

这就是 strong wolfe condition，为何这样做？是因为上文注解处提到的：$\Phi'(\alpha)$ 甚至可能大于 0。所以为了避免 $\Phi'(\alpha)$ 变为正数后过大，引出了强条件，如下图所示：

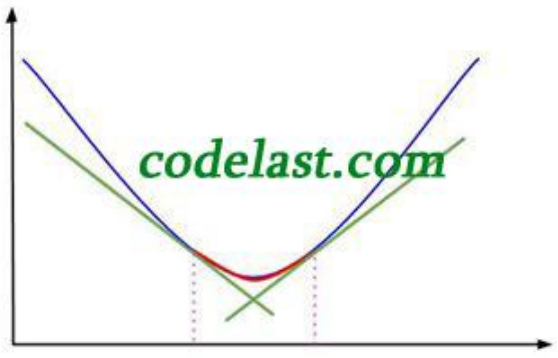


图 3.5 why strong wolfe condition

可以看到，强条件中图中红线被巧妙地“夹击”了。

3.2 Interpolation

所谓插值法的概念就是在一个坐标系内有点 x_0、x_1、…、x_k 以及所对应的纵坐标 y_0、y_1、…、y_k，找到一个函数 $P_n(x_i)$（通常是多项式函数）穿过这些点，满足 $P_n(x_i)=f(x_i)$，在更高级的 hermite 插值中，不仅要满足 $P_n(x_i)=f(x_i)$，还要满足 $P'_n(x_i)=f'(x_i)$，即导数值也要一致，如下图所示：

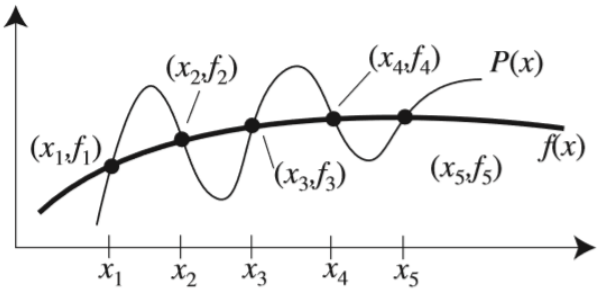


图 3.6 不满足导数值相等的插值法

在 Line search 技术中，插值法的作用有两个：

1. **切换区间**：在寻找满足 strong wolfe condition 的区间过程中，不断去搜寻 α 间隔，如果当前间隔不满足 strong wolfe condition，则继续向前搜索下一个间隔，这切换间隔的过程使用了二次\三次插值法（或 Hermite Interpolation）找出 α_{new}，区间$[\alpha_{i-1}, \alpha_i]$就切换为了$[\alpha_i, \alpha_{new}]$。
2. **区间内搜索最小值**：在找到满足 strong wolfe condition 的区间$[\alpha_{i-1}, \alpha_i]$之后，使用插值法找出满足两端点$[\alpha_{i-1}, \alpha_i]$的多项式 $P_n(x_i)$，再通过求导=0 这种解析的方法找出 $P_n(x_i)$ 的极值点 α_{new}，用该点替换掉 α_{i-1}, α_i 的**其中一个，然后缩小区间，继续寻找最小值。**

3.2.1 Quadratic Interpolation

现在假设我们 line search 第一次尝试的 $\alpha=\alpha_0$，所以要寻找的区间是$[0, \alpha_0]$，要在这个区间上找出满足 $\Phi(\alpha)$极值 α_1。我们从二次插值（quadratic）开始推导。

我们先用一元二次函数 $\Phi_q(\alpha)$来做插值，这个二次函数 $\Phi_q(\alpha)$需要满足以下条件：[9]

$$\begin{cases} \Phi(0)=\Phi_q(0) \\ \Phi'(0)=\Phi_q'(0) \\ \Phi(\alpha_0)=\Phi_q(\alpha_0) \end{cases}$$

其中 $\Phi_q(\alpha)=a\cdot\alpha^2+b\cdot\alpha+c$

因此由方程组可以得到：

$$\begin{cases} a=\Phi(0) \\ b=\Phi'(0) \\ c=(\Phi(\alpha_0)-\Phi(0)-\Phi'(0)\cdot\alpha_0)/\alpha_0^2 \end{cases}$$

由于一元二次函数的极小值出现在 $\alpha=-b/2a$，所以

[9] （猜测）考虑到在牛顿迭代法中，导数 $\Phi'(0)$的值从上一轮迭代中已经计算得到，可以直接使用

$$\alpha_1 = -\frac{b}{2a} = \frac{\alpha_0^2 \Phi^{'}(0)}{2[\Phi(0) + \alpha_0 \cdot \Phi^{'}(0) - \Phi(\alpha_0)]}$$

迭代缩短区间：求得 Φ (α1)，并和 Φ (α0)的（原区间端点）的函数值比较，取其较小者，作为新的 α1，然后以此点作为新的区间端点[0, α1]，删去多余部分。如下图所示：

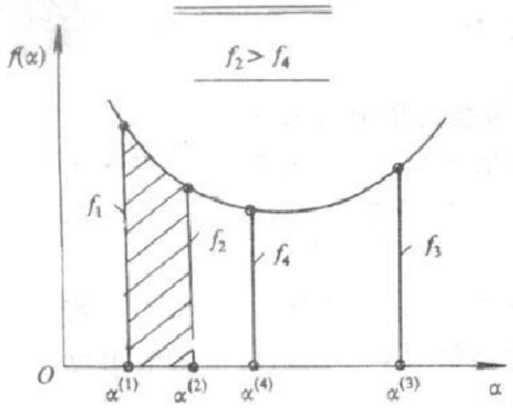


图 3.7 阴影部分是删除的区间部分

3.2.2 Cubic Interpolation

但是，在第一次迭代后，我们有了 Φ (0)、Φ ′(0)、Φ (α0)、Φ (α1)的值，根据这些值我们可以用三次插值法更精确地拟合曲线。三次插值函数设为$\Phi_c(\alpha) = a \cdot \alpha^3 + b \cdot \alpha^2 + c \cdot \alpha + d$

$$\begin{cases} \Phi(0) = \Phi_c(0) \\ \Phi^{'}(0) = \Phi_c^{'}(0) \\ \Phi(\alpha_0) = \Phi_c(\alpha_0) \\ \Phi(\alpha_1) = \Phi_c(\alpha_1) \end{cases}$$ 由方程组的第 1 个和第 2 个可得 c=Φ′ (0)，d=Φ (0)

所以式子变为$\Phi_c(\alpha) = a \cdot \alpha^3 + b \cdot \alpha^2 + \Phi^{'}(0) \cdot \alpha + \Phi(0)$

现在问题转变为求 a 和 b 两个系数：

$$\begin{cases} \Phi_c(\alpha_0) = \Phi(\alpha_0) \\ \Phi_c(\alpha_1) = \Phi(\alpha_1) \end{cases} => \begin{cases} a \cdot \alpha_0^3 + b \cdot \alpha_0^2 + \alpha_0 \cdot \Phi^{'}(0) + \Phi(0) = \Phi(\alpha_0) \\ a \cdot \alpha_1^3 + b \cdot \alpha_1^2 + \alpha_1 \cdot \Phi^{'}(0) + \Phi(0) = \Phi(\alpha_1) \end{cases}$$

将等式左边的后两项移到等式右边，就得到如下的线性方程组：

$$\begin{bmatrix} \alpha_0^3 & \alpha_0^2 \\ \alpha_1^3 & \alpha_1^2 \end{bmatrix} \bullet \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \Phi(\alpha_0) - \Phi(0) - \alpha_0 \cdot \Phi^{'}(0) \\ \Phi(\alpha_1) - \Phi(0) - \alpha_1 \cdot \Phi^{'}(0) \end{bmatrix}$$

然后将方程左右两边同时左乘$\begin{bmatrix} \alpha_0^3 & \alpha_0^2 \\ \alpha_1^3 & \alpha_1^2 \end{bmatrix}^{-1}$ 即可得到最后的 a、b 值，由于

$A^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$，所以最终：

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\alpha_0^3\alpha_1^2 - \alpha_1^3\alpha_0^2} \begin{bmatrix} \alpha_1^2 & -\alpha_0^2 \\ -\alpha_1^3 & \alpha_0^3 \end{bmatrix} \bullet \begin{bmatrix} \Phi(\alpha_0) - \Phi(0) - \alpha_0 \cdot \Phi'(0) \\ \Phi(\alpha_1) - \Phi(0) - \alpha_1 \cdot \Phi'(0) \end{bmatrix}$$

求出 a、b、c、d 所有系数后对$\Phi_c(\alpha) = a \cdot \alpha^3 + b \cdot \alpha^2 + c \cdot \alpha + d$求导使之=0，则

$\Phi_c'(\alpha) = 3a \cdot \alpha^2 + 2b \cdot \alpha + c = 0$，由于一元二次方程的根$\alpha = \frac{-b \pm \sqrt{b^2 - 3a\Phi'(0)}}{3a}$，又因为我们希望取值$\alpha_2 \in [0, \alpha_1]$，所以需要 α>0，故$\alpha_2 = \frac{-b + \sqrt{b^2 - 3a\Phi'(0)}}{3a}$。

将上述过程中的 α_0、α_1 替换为 α_{i-1}、α_i（也就是每次最新的两个 Φ 值），同样的公式就估计了下次迭代的 α_{i+1} 的值，不断重复此过程，在$\alpha_{i+1} \in [0, \alpha_i]$中找到令 Φ(α)最小的值。

注意：

如果 α_i 比 α_{i-1} 小太多，**或者** α_{i-1} 和 α_i 太接近时，$\alpha_i = \alpha_{i-1}/2$，该保护措施(safeguards)保证下一次的步长不至于太小，因为：

(1) $\alpha_{i+1} \in [0, \alpha_i]$，所以当 α_i 很小时，α_{i+1} 也很小。

(2)当 α_{i-1} 与 α_i 太靠近时有 a≈b≈∞，根据 α_{i+1} 表达式可知 $\alpha_{i+1} \approx 0$

如果 Φ ′(α_i)这个导数的值不是很难求，在计算 Φ (α_i)的执行过程中可以“顺便”计算出来，那么更近似的插值法是 Hermite 插值法[10]，由于我们的所讲程序中没有用到 Hermite 插值法，因此该内容放到附录里。

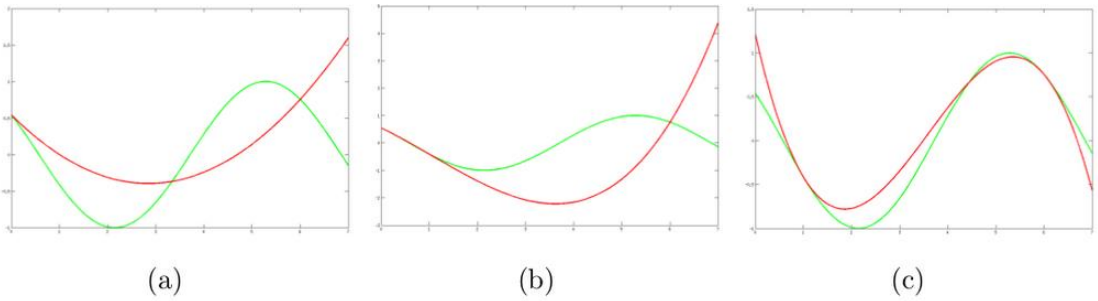


图 3.8 (a) 二次插值(quadratic) (b)三次插值(cubic) (c)Hermite 插值

3.3 Φ(α) and Φ′(α) in LR

看到上一节里，经常要求在某个点 α_i 处的 Φ(α_i)，回到 LR 的似然函数

$$l(\theta) = logL(\theta) = \sum_{i=1}^{m} (y^{(i)} log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

我们希望能快速得到 Φ(α) = f(x_k+α_k P_k)，但是由于$h_\theta(x^{(i)})$内包含 $x^{(i)}$和θ的内积，因此求该$l(\theta)$需要经过外层 m 次循环，和内层 n 次的循环，时间复杂度为 m * n。

[10]埃尔米特（Charles Hermite，1822—1901）法国数学家：是一位数学考试经常不及格的数学家

如果我们每次代入一个 α_i 就要 m * n 次循环就显得太慢了，我们希望能加快速度。这里设 $d=\alpha_k P_k$。

$$\Phi(\alpha)=l(\theta+d)=\log L(\theta+d)=\sum_{i=1}^{m}(y^{(i)}\log h_{\theta+d}(x^{(i)})+(1-y^{(i)})\log(1-h_{\theta+d}(x^{(i)}))$$

$$\text{代入}h_{\theta+d}(x^{(i)})=\frac{1}{1+e^{-(\theta+d)^T x^{(i)}}}\underset{\text{省去第i行编号}}{=}\frac{1}{1+e^{-((\theta_1+d)x_1+(\theta_2+d)x_2+\ldots+(\theta_n+d)x_n)}}=\frac{1}{1+e^{-(\theta_1x_1+\theta_2x_2+\ldots+\theta_nx_n+d\cdot(x_1+x_2+\ldots x_n))}}$$

$$=\frac{1}{1+e^{-(\theta^T x+d\sum_{j=1}^{n}x_j)}}=\frac{1}{1+\underbrace{e^{-(\theta^T x)}}_{\text{存下来}}\bullet\left(\underbrace{e^{-\sum_{j=1}^{n}x_j}}_{\text{存下来}}\right)^d}$$

这样需要将每一行的 $e^{-(\theta^T x^{(i)})}$ 和 $e^{-\sum_{j=1}^{n}x_j^{(i)}}$ 这两个数字存下来，以便下次调用 Φ(α)的时候可以直接使用，不用重复内部循环，但是外部的 m 次循环不可避免。

Φ(α)空间复杂度：额外耗费内存 2 * m（可重复使用这部分存储内容）；

Φ(α)时间复杂度 m。

$$\Phi'(\alpha)=\frac{d\Phi(\alpha)}{d\alpha}=\nabla f(x_k+\alpha_k P_k)^T\cdot P_k$$

在 LR 中 $\nabla f(\theta_k+\alpha_k P_k)=\begin{bmatrix}\sum_{i=1}^{m}(x_1^{(i)}\bullet(y^{(i)}-h_{\theta_k+\alpha_k P_k}(x^{(i)})))\\ \sum_{i=1}^{m}(x_2^{(i)}\bullet(y^{(i)}-h_{\theta_k+\alpha_k P_k}(x^{(i)})))\\ \cdots\\ \sum_{i=1}^{m}(x_n^{(i)}\bullet(y^{(i)}-h_{\theta_k+\alpha_k P_k}(x^{(i)})))\end{bmatrix}$，这是 n X 1 的向量，转置后为 1 X n；而 P_k 在拟牛顿迭代法中是 n X 1 的向量[11]，所以最终 $\Phi'(\alpha)$ 是 1 个数字。

每次传入不同 α，Φ′(α)变化的部分仅为 $h_{\theta_k+\alpha_k P_k}(x^{(i)})$，这里我们同样可以将 $e^{-(\theta^T x^{(i)})}$ 和 $e^{-\sum_{j=1}^{n}x_j^{(i)}}$ 这两个数字存下来，另外 P_k 不变，所以存储一个 n X 1 的向量即可。

Φ′(α)空间复杂度: 2 * m + n（可重复使用这部分存储内容）。

Φ′(α)时间复杂度: m*n + n，后一次 n 次循环为计算那两个向量 $\nabla f(x_k+\alpha_k P_k)^T\cdot P_k$ 的乘法。

[11] 牛顿法中的 $-H^{-1}(\theta^{(n)})\cdot\nabla f(\theta^{(n)})$

Chapter 4. 牛顿迭代法(Newton-Raphson method)

3.1 Taylor series to Newton

从最简单的例子开始，假设我们有一个单变量函数 f(θ)，我们想找到让 f(θ)取全局最小值的变量θ^*。假设函数 f(θ)是平滑的(smooth)且θ^*是一个 regular interior minimum，即表示函数在θ^*点的导数为零且二阶导数为正。在全局最小值附近我们可以对函数 f(θ)做泰勒展开：

$$f(\theta) \approx f(\theta^*) + \frac{1}{2}(\theta - \theta^*)^2 \frac{d^2 f}{d\theta^2}\bigg|_{\theta=\theta^*}$$

这样，最小化函数 f(θ)的值就转化为最小化上式第二部分二项式的值。

我们思路是：每次迭代寻找到的下一个$\theta^{(n+1)}$是在当前$\theta^{(n)}$附近的极小值，也即当前$\theta^{(n)}$附近泰勒展开式对$\theta^{(n+1)}$求导=0（即$\boldsymbol{\nabla} f\left(\theta^{(n+1)}\right) = 0$）。

我们猜测一个初始值$\theta^{(0)}$并在该数值附近做二级泰勒展开，即：

$$f(\theta) \approx f\left(\theta^{(0)}\right) + (\theta - \theta^{(0)}) \frac{df}{d\theta}\bigg|_{\theta=\theta^{(0)}} + \frac{1}{2}(\theta - \theta^{(0)})^2 \frac{d^2 f}{d\theta^2}\bigg|_{\theta=\theta^{(0)}}$$

对上式右端的 θ 求导等于零，设其解为 $\theta^{(1)}$，则有：

$$f'(\theta^{(0)}) + \frac{1}{2} f''(\theta^{(0)}) 2(\theta^{(1)} - \theta^{(0)}) = 0$$

注意求导时，第一项由于不含 θ 被约去，第二项中的$\theta^{(0)}$也被约去，进一步整理可得：

$$\theta^{(1)} = \theta^{(0)} - \frac{f'(\theta^{(0)})}{f''(\theta^{(0)})}$$

$\theta^{(1)}$相比于 $\theta^{(0)}$是对于 θ^*的一个更好的预估值。我们循环以上过程，即：

$$\theta^{(n+1)} = \theta^{(n)} - \frac{f'(\theta^{(n)})}{f''(\theta^{(n)})}$$

就可以不断逼近于 θ^*。

当函数是一个多变量函数时，即 $f(\theta_1, \theta_2, \ldots, \theta_p)$，牛顿迭代按照下式计算：

$$\theta^{(n+1)} = \theta^{(n)} - H^{-1}(\theta^{(n)}) \cdot \boldsymbol{\nabla} f\left(\theta^{(n)}\right)$$

其中$\boldsymbol{\nabla}$f 是函数 f 的梯度，即为$[\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \cdots, \frac{\partial f}{\partial \theta_p}]$。$H$ 为函数 f 的 Hessian 矩阵，即$H_{ij} = \partial^2 f / \partial\theta_i \partial\theta_j$。其中求 H 的逆矩阵比较困难，可以通过将 H 近似为对角矩阵来计算。

在牛顿迭代法中，$H^{-1}(\theta^{(n)}) \cdot \boldsymbol{\nabla} f\left(\theta^{(n)}\right)$决定着搜索方向，通常还要加上步长 λ（小正数），由一维搜索确定每一步走多远。

3.2 Mathematical background

Hessian 矩阵的一般表示如下：

$$H=\begin{bmatrix} \frac{\partial^2 f}{\partial\theta_1^2} & \frac{\partial^2 f}{\partial\theta_1\partial\theta_2} & \cdots & \frac{\partial^2 f}{\partial\theta_1\partial\theta_n} \\ \frac{\partial^2 f}{\partial\theta_2\partial\theta_1} & \frac{\partial^2 f}{\partial\theta_2^2} & \cdots & \frac{\partial^2 f}{\partial\theta_2\partial\theta_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 f}{\partial\theta_n\partial\theta_1} & \frac{\partial^2 f}{\partial\theta_n\partial\theta_2} & \cdots & \frac{\partial^2 f}{\partial\theta_n^2} \end{bmatrix}$$

由于二阶偏导数交换次序没有区别，即 $\frac{\partial}{\partial\theta_2}(\frac{\partial f}{\partial\theta_1})=\frac{\partial}{\partial\theta_1}(\frac{\partial f}{\partial\theta_2})$，所以 hessian 矩阵是对称矩阵。

Hessian 矩阵的作用：类似于求解一维函数的极小值的判定方法。

复习一下一维函数的极值判定方法：

设 f(x)在 x_0 二阶可导，且 $f'(x_0)=0$, $f''(x_0)\neq 0$

(1)若 $f''(x_0)<0$，则 f(x)在 x_0 处取得极大值。

(2)若 $f''(x_0)>0$，则 f(x)在 x_0 处取得极小值。

类似地，多元函数的极值判定方法：

$f(X)=f(x_1,x_2,...,x_n)$ 在 X= $(x_1,x_2,...,x_n)$ 的某个邻域内有连续的一阶偏导数(gradient)和二阶偏导数(hessian matrix)，且一阶偏导数=0，如下联立方程组：

$$\nabla f(X)=0 \Rightarrow \begin{cases} \frac{\partial f}{\partial x_1}=0 \\ \frac{\partial f}{\partial x_2}=0 \\ \cdots \\ \frac{\partial f}{\partial x_n}=0 \end{cases}$$ 求得驻点：$M_i=(x_1,x_2,...,x_n)$

(1) 如果 H(M_i)是**正定**矩阵，则 f(M_i)取得局部**极小值**。

(2) 如果 H(M_i)是**负定**矩阵，则 f(M_i)取得局部**极大值**。

(3) 如果 H(M_i)是不定矩阵，则 f(M_i)不是极值。

再补充一个正定矩阵(positive definite matrix)的概念：

设 M 是 n 阶**对称**矩阵，如果对**任意**非零向量 $\vec{z}$，$\vec{z}^T \bullet M \bullet \vec{z}>0$，其中 $\vec{z}^T$ 表示 $\vec{z}$ 的转置，就称 M 是正定矩阵。正定矩阵可以通过赫尔维茨定理(Hurwitz theorem)来判定。

举个最简单的正定矩阵的例子，单位矩阵(Identity Matrix)：

$$I=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix},\text{设}\vec{z}=\begin{bmatrix}1&2&3\end{bmatrix}$$

$$\vec{z}^{T}\bullet I\bullet\vec{z}=\begin{bmatrix}1&2&3\end{bmatrix}\bullet\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\bullet\begin{bmatrix}1\\2\\3\end{bmatrix}$$

$$=\begin{bmatrix}1&2&3\end{bmatrix}\bullet\begin{bmatrix}1\\2\\3\end{bmatrix}=1^2+2^2+3^2=13>0$$

（不要求掌握）来试试用赫尔维茨定理判定单位矩阵：

赫尔维茨定理：对称矩阵 A 为正定矩阵的充分必要条件是：A 的各阶主子式都为正。

即：$a_{11}>0,\begin{vmatrix}a_{11}&a_{12}\\a_{21}&a_{22}\end{vmatrix}>0,\ldots,\begin{vmatrix}a_{11}&\ldots&a_{1n}\\\ldots&\ldots&\ldots\\a_{n1}&\ldots&a_{nn}\end{vmatrix}>0$

在单位矩阵 $\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$ 中，$\Delta_1=1>0$，$\Delta_2=\begin{vmatrix}1&0\\0&1\end{vmatrix}=1>0$，

$\Delta_3=\begin{vmatrix}1&0&0\\0&1&0\\0&0&1\end{vmatrix}=1>0$，

从而判定单位矩阵都是正定矩阵。

*补充概念：**主子式的概念**如下。*

n 阶行列式的 i 阶主子式为：

在 n 阶行列式中，任选 i 行（假设 i=3 阶，选取 1、3、7 行时），再选取相同行号的列（1、3、7 列），

由上述选取的行列交汇处的元素所组成的新的行列式就称为“n 阶行列式的一个 i 阶主子式”。

***特殊的：**n 阶行列式的 i 阶**顺序**主子式：*

上述 i 阶主子式中定义中，由 1~i 行和 1~i 列所确定的子式即为“n 阶行列式的 i 阶顺序主子式”。

例如：

1 阶时：取第 1 行，第 1 列

2 阶时：取第 1、2 行，第 1、2 列

3 阶时：取第 1、2、3 行，第 1、2、3 列

4 阶时：取第 1、2、3、4 行，第 1、2、3、4 列

值得注意的是，根据定义，i 阶主子式是不唯一的，而 i 阶顺序主子式是唯一的。

*正定矩阵的**两个性质**：(1)正定矩阵必然可逆。(2)正定矩阵的逆矩阵还是正定矩阵。*[12]

3.3 Hessian Matrix in LR

现在来求 Logistic Regression 中的 Hessian 矩阵是什么样的，假设 Hessian Matrix 的第 k 行第 r 列=$\frac{\partial^2 f}{\partial\theta_k\partial\theta_r}$

$$\frac{\partial^2\ln(\theta)}{\partial\theta_k\partial\theta_r}=\frac{\partial(\sum_{i=1}^{m}x_k^{(i)}\bullet(y^{(i)}-h_\theta(x^{(i)})))}{\partial\theta_r}$$

其中$h_\theta(x^{(i)})=\frac{1}{1+e^{-\theta^T x}}$

$$\frac{\partial^2\ln(\theta)}{\partial\theta_k\partial\theta_r}=\frac{\partial(\sum_{i=1}^{m}x_k^{(i)}\bullet(y^{(i)}-\frac{1}{1+e^{-\theta^T x}}))}{\partial\theta_r}$$

$$=\frac{\partial(\sum_{i=1}^{m}(\overbrace{x_k^{(i)}\bullet y^{(i)}}^{\text{对}\theta_r\text{求导=0}}-x_k^{(i)}\bullet\frac{1}{1+e^{-\theta^T x}}))}{\partial\theta_r}$$

$$=\frac{-\partial\sum_{i=1}^{m}(x_k^{(i)}\bullet\frac{1}{1+e^{-\theta^T x}})}{\partial\theta_r}$$

由于$\frac{\partial(h_\theta(x^{(i)}))}{\partial\theta_r}\overbrace{=}^{chain\ rule}h_\theta(x^{(i)})\bullet(1-h_\theta(x^{(i)}))\bullet x_r^{(i)}$

所以$\frac{\partial^2\ln(\theta)}{\partial\theta_k\partial\theta_r}=-\sum_{i=1}^{m}(x_k^{(i)}\bullet x_r^{(i)}h_\theta(x^{(i)})\bullet(1-h_\theta(x^{(i)})))=\sum_{i=1}^{m}(x_k^{(i)}\bullet x_r^{(i)}h_\theta(x^{(i)})\bullet(h_\theta(x^{(i)})-1))$

所以 Logistic Regression 中的 Hessian 矩阵（n X n）如下：

$$H=\begin{bmatrix}-\sum_{i=1}^{m}x_1^{(i)}x_1^{(i)}\bullet\underbrace{h_\theta(x^{(i)})\bullet(1-h_\theta(x^{(i)}))}_{v_\theta(x^{(i)})} & -\sum_{i=1}^{m}x_1^{(i)}x_2^{(i)}\bullet v_\theta(x^{(i)}) & \dots & -\sum_{i=1}^{m}x_1^{(i)}x_n^{(i)}\bullet v_\theta(x^{(i)})\\ -\sum_{i=1}^{m}x_2^{(i)}x_1^{(i)}\bullet v_\theta(x^{(i)}) & -\sum_{i=1}^{m}x_2^{(i)}x_2^{(i)}\bullet v_\theta(x^{(i)}) & \dots & -\sum_{i=1}^{m}x_2^{(i)}x_n^{(i)}\bullet v_\theta(x^{(i)})\\ \dots & \dots & \dots & \dots\\ -\sum_{i=1}^{m}x_n^{(i)}x_1^{(i)}\bullet v_\theta(x^{(i)}) & -\sum_{i=1}^{m}x_n^{(i)}x_2^{(i)}\bullet v_\theta(x^{(i)}) & \dots & -\sum_{i=1}^{m}x_n^{(i)}x_n^{(i)}\bullet v_\theta(x^{(i)})\end{bmatrix}$$

（不要求掌握）**对角化理论**

[12] (1): 正定的充分必要条件是其顺序主子式全大于 0，若 A 正定,必有 |A|>0，故 A 可逆.
(2): $Z^TA^{-1}Z=Z^TA^{-1}AA^{-1}Z=(A^{-1}Z)^TA(A^{-1}Z)>0$，所以 A^{-1} 正定

如果你对线性代数中的对角化理论较为了解，注意这个 LR 中的 Hessian 矩阵可以由

$$H = \vec{x}^{T} \bullet v \bullet \vec{x}$$

其中 v 为对角矩阵(diagonal matrix)，共 m 行 m 列，对角线上 m 个元素，每个元素=$v_\theta(x^{(i)})$

$$V = \begin{bmatrix} v_\theta(x^{(1)}) & 0 & \dots & 0 \\ 0 & v_\theta(x^{(2)}) & \dots & 0 \\ \dots & 0 & \dots & \dots \\ 0 & 0 & \dots & v_\theta(x^{(n)}) \end{bmatrix}$$

X 矩阵为 m 行 n 列的矩阵，X^T 为 X 矩阵的转置矩阵。

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \dots & \dots & \dots & \dots \\ x_1^{(m)} & x_2^{(m)} & \dots & x_n^{(m)} \end{bmatrix} = \begin{bmatrix} \overrightarrow{x^{(1)}}^T \\ \overrightarrow{x^{(2)}}^T \\ \dots \\ \overrightarrow{x^{(m)}}^T \end{bmatrix}$$

矩阵求逆[13]

LR 中的一个步骤是要计算矩阵的逆矩阵$inv(X^T W^{(n)} X)$。在求解逆矩阵的过程中一般采用 Doolittle's method 来计算。设要计算 A 矩阵的逆矩阵 B = A^{-1}，首先利用 Doolittle's method 将 A 矩阵分解成 L 和 U 两个下/上三角矩阵，再利用 forward-substitution 和 back-substitution 计算出逆矩阵 B。示例如下。

设：

$$A = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} = L \times U$$

其中 L 是一个下三角矩阵，U 是一个上三角矩阵。首先设：

$$L \times U = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} = \mathrm{A}$$

首先用 L 的第一行乘以 U 的第 1 至第 3 列，得到

$$U_{11} = 25;\ U_{12} = 5;\ U_{13} = 1$$

$$L \times U = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} 25 & 5 & 1 \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} = \mathrm{A}$$

接着用 L 的第 2 至第 3 行乘以 U 的第 1 列，得到

$$L_{21} = \frac{64}{25} = 2.56;\ L_{31} = \frac{144}{25} = 5.76$$

$$L \times U = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & L_{32} & 1 \end{bmatrix} \begin{bmatrix} 25 & 5 & 1 \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} = \mathrm{A}$$

接着用 L 的第 2 行乘以 U 的第 2 和第 3 列，得到

[13] 实践中，诸如 python 的 numpy 等矩阵库已经实现了比较高效的矩阵求逆算法

$$U_{22} = 8 - (2.56 \times 5) = -4.8;\ \ U_{23} = 1 - (2.56 \times 1) = -1.56$$

$$L \times U = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & L_{32} & 1 \end{bmatrix} \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & U_{33} \end{bmatrix} = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} = \mathrm{A}$$

接着用 L 的第 3 行乘以 U 的第 2 列，得到

$$L_{32} = \frac{12 - (5.76 \times 5)}{-4.8} = 3.5$$

$$L \times U = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix} \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & U_{33} \end{bmatrix} = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} = \mathrm{A}$$

接着用 L 的第 3 行乘以 U 的第 3 列，得到

$$U_{33} = 1 - 5.76 - 3.5 \times (-1.56) = 0.7$$

$$L \times U = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix} \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix} = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} = \mathrm{A}$$

因为 B 是 A 的逆矩阵，所以有：

$$A \times B = L \times U \times B = \mathrm{E}$$

其中 E 为单位对角矩阵。设$Z = U \times B$，则有：

$$A \times B = L \times Z = \mathrm{E}$$

对于单位矩阵的第一列，可以得到

$$\begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix} \begin{bmatrix} Z_{11} \\ Z_{21} \\ Z_{31} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

可以计算出：

$$Z_{11} = 1;\ \ Z_{21} = -2.56;\ \ Z_{31} = 3.2$$

这一步计算被称为 forward-substitution。

又因$Z = U \times B$，即：

$$\begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix} \begin{bmatrix} B_{11} \\ B_{21} \\ B_{31} \end{bmatrix} = \begin{bmatrix} 1 \\ -2.56 \\ 3.2 \end{bmatrix}$$

可以计算得到：

$$B_{11} = 0.0476;\ \ B_{21} = -0.952;\ \ B_{31} = 4.571$$

这一步计算被称为 back-substitution。

这样就得到了 A 的逆矩阵 B 的第一列。同理可以得到其余的两列。

3.4 Implement

一种近似的牛顿迭代算法用下式来迭代求得 θ 的值：

$$\theta^{(n+1)} = \theta^{(n)} + inv(X^T W^{(n)} X) X^T (Y - \theta^{(n)})$$

其中 X 被称为 design matrix，其第一列全为 1.0，X_{ij} 表示第 i 个样本的第 j 个属性的值，其行数等于样本数，列数等于特征数量加 1。X^T 表示矩阵 X 的转置。Y 为样本的分类结果（在 LR 中为二分值 0 或 1）。W 被称为 weight matrix，是一个 m * m 的对角矩阵，其中 m

为 X 矩阵的行数（样本数）。其对角线上的每个值等于该行相应样本的 LR 概率 p 乘以(1 - p)。

在实际计算中并不需要显示地算出 W 矩阵，而是直接通过 p 和 X 一并算出 $W^{(n)}X$。

注意：

1. W 矩阵有可能不存在逆矩阵，需要特殊处理这种情况。
2. 初始化 θ 向量可将其设置为全 0 的向量。

例子：

The theta update equation and the W matrix are best explained with a concrete example. Suppose for simplicity that the training set consists only of the first five lines of data shown in Figure 1. So the design matrix X would be:

1.00	48.00	1.00	4.40
1.00	60.00	0.00	7.89
1.00	51.00	0.00	3.48
1.00	66.00	0.00	8.41
1.00	40.00	1.00	3.05

The Y dependent variable vector would be:

0
1
0
1
0

Let’s assume that the old beta vector of values to be updated, b[t-1], is:

1.00
0.01
0.01
0.01

With these values for X and beta, the old p vector, p[t-1], is:

0.8226
0.8428
0.8242
0.8512
0.8085

Notice that if we presume p values < 0.5 are interpreted as y = 0 and p values >= 0.5 are interpreted as y = 1, the old beta values would correctly predict only two out of five cases in the training data.

The weights matrix W is an m x m matrix where m is the number of rows of X. All the values in the W matrix are 0.0 except for those m values that are on the main diagonal. Each of these values equals the corresponding p value multiplied by 1-p. So for this example, W would be size 5 x 5. The upper-left cell at [0,0] would be (0.8226)(1 - 0.8226) = 0.1459. The cell at [1,1] would be (0.8428)(1 - 0.8428) = 0.1325, and so on. The quantity (p)(1-p) represents the calculus derivative of the sigmoid function.

In practice, the W matrix is not computed explicitly because its size could be huge. If you had 1,000 rows of training data, matrix W would have 1,000,000 cells. Notice the beta update equation has a term W[t-1]X, which means the matrix product of W[t-1] and X. Because most of the values in W[t-1] are zero, most of the matrix multiplication terms are also zero. This allows W[t-1] times X to be computed directly from p[t-1] and X, without explicitly constructing W. Several of the math references that describe IRLS with the NR algorithm for LR use the symbol X~ (X tilde) for the product of W[t-1] and X. See method ComputeXtilde in the code download for implementation details.

LR 的一个很大的缺陷是循环迭代了多次后，会造成对于训练样本的过拟合(over-fitting)。很难确定 LR 循环迭代多少次比较合适。停止迭代的几种条件如下：

1. 每次迭代后，计算出各个样本的概率 p，并和已知的结果 Y 做比较，计算 Mean Square Error(MSE)。当 MSE 小于某个设定值时退出循环。
2. 设定一个最大循环次数，当循环次数超过该数值时结束循环。循环过程中当求解逆矩阵失败时也可结束循环。另一种方式是在跳出循环后对得到的参数 θ 和上一次循环得到的参数做平均，并作为结果返回。
3. 检查每个参数的变化情况。当所有参数的变化值小于一个设定值时跳出循环。此时很容易出现过拟合的情况。可以选择返回之前循环所得到的参数值。
4. 当某一个参数的两次循环结果的变化超过某一个设定的阀值时（例如 1000），表明此时迭代已失去了控制，需要抛出一个异常而不是返回最近的参数值。
5. 当某次循环后得到的参数用于分类，当某一个样本的误差是上一次循环误差的 4 倍以上时，需要跳出循环并返回参数值。

使用 Newton-Raphson 算法来估计参数时需要做实验来调整迭代停止条件，达到最优。

Chapter 4 拟牛顿迭代法(Quasi-Newton Methods)

由于牛顿迭代法的需要求 hessian 矩阵的逆矩阵，这里会产生几个问题：

1. **逆矩阵**：不是所有矩阵都有逆矩阵，只有非奇异矩阵（或称满秩矩阵）才有逆矩阵。[14][15]
2. **耗时**：牛顿迭代法中的求逆矩阵复杂度较高，非常耗时。
3. **Hessian 正定**：牛顿迭代法的效果与初值 x_0 的选择有密切关系，如果选择不好，造成 Hessian 矩阵不为正定矩阵时牛顿迭代法可能朝着错误的方向迭代。而且在迭代过程中很难保证 Hessian 矩阵一直为正定矩阵（尤其是在远离极小值处，Hessian 矩阵一般不能正定）。

拟牛顿迭代法的**基本思想**就是从这个 Hessian 矩阵入手，用一种近似（approximate）的矩阵来代替 Hessian 矩阵，这样每次迭代时更新这个近似矩阵，最终达到与牛顿迭代法搜索方向大体一致的效果。

4.1 secant equation

首先，我们从上一章的提到过的多元函数的泰勒展开公式开始：

$$f(\theta) = f(\theta^{(k)}) + g_k^T \bullet (\theta - \theta^{(k)}) + \frac{1}{2}(\theta - \theta^{(k)})^T H(\theta^{(k)})(\theta - \theta^{(k)})$$

两边同时对 θ 求导（梯度）可得：

$$\nabla f(\theta) = g_k + H(\theta^{(k)}) \bullet (\theta - \theta^{(k)})$$

利用极小点必要条件令 $\nabla f(\theta) = 0$ 即可得到：

$$\theta^{(k+1)} = \theta^{(k)} \underbrace{-\mathrm{H}_k^{-1} g_k}_{p_k} = \theta^{(k)} + \alpha \bullet p_k \quad \text{加上一维搜索的步长}\alpha$$

明确几个公式中用到的符号：

θ : n X 1 的向量，即估计函数中的自变量 x，有时候用 x 表达相同的含义（最优化）
B_k : Hessian 矩阵的近似矩阵于第 k 轮迭代，LR 中 n X n 维
D_k : H^{-1},Hessian 矩阵的**逆矩阵**的近似矩阵于第 k 轮迭代，LR 中 n X n 维
g_k : f 的梯度，即 $g_k = g(\theta^{(k)}) = \nabla f(\theta^{(k)})$ ，LR 中 n X 1 维
p_k : 搜索方向，即 $-\mathrm{H}_k^{-1} g_k$ ，LR 中 n X 1 维
α : 迭代步长，一个小正数，靠一维搜索(line search)确定
$y_k = g_{k+1} - g_k$ 两次迭代间梯度的变化量，LR 中 n X 1 维
$s_k = \theta^{(k+1)} - \theta^{(k)}$ 两次迭代间自变量 θ(即一般方程的 x)的变化量，LR 中 n X 1 维

[14] 对称矩阵不一定可逆，比如当矩阵所有元素都为 0 时此矩阵也是对称矩阵，但不可逆

[15] 对称矩阵若可逆，则其逆矩阵也是对称矩阵 $A^T=A$; $(A^{-1})^T = (A^T)^{-1} = A^{-1}$; 所以 A^{-1} 是对称矩阵

在上面这个式子取 $\theta = \theta^{(k+1)}$，可得：

$$g_{k+1} = g_k + H(\theta^{(k)})\bullet(\theta^{(k+1)} - \theta^{(k)})$$

整理后得到：

$$\underbrace{g_{k+1} - g_k}_{y_k} = \underbrace{H(\theta^{(k)})}_{H_k}\bullet\underbrace{(\theta^{(k+1)} - \theta^{(k)})}_{s_k}$$

$$y_k = \underbrace{H(\theta^{(k)})}_{\text{用}B_k\text{模拟}}\bullet s_k$$

或 $\underbrace{H(\theta^{(k)})^{-1}}_{\text{用}D_k\text{模拟}}\bullet \mathrm{y}_k = \mathrm{s}_k$

这个方程就是割线方程（secant equation），割线方程就是**拟牛顿条件**，模拟矩阵 B_k、D_k 只有满足这个条件才能称之为 approximate matrix。

在上一章里，我们说明了 Hessian 正定与多元函数极小值的关系，下面来从另一个角度来求证这一点：

求证：如果 H_k 是正定的（这就意味着 H_k^{-1} 也是正定）那么就可以保证牛顿搜索方向是下降的。

证明：

$$\mathrm{x}^{(k+1)} = x^{(k)} - \alpha H_k^{-1}\cdot g_k$$

这里的 α 是代表步长的小正数，即每一次走多远，由一维搜索(line search)确定。[16]

代入(4.1)式，可得

$$\mathrm{f}(\mathrm{x}^{(k+1)}) = f(x^{(k)} - \lambda H_k^{-1}\cdot g_k)$$

$$= f(x^{(k)}) + g_k^T\cdot(x^{(k)} - \lambda H_k^{-1}\cdot g_k - x^{(k)}) + \frac{1}{2}(x^{(k)} - \lambda H_k^{-1}\cdot g_k - x^{(k)})^T\cdot H_k\cdot(x^{(k)} - \lambda H_k^{-1}\cdot g_k - x^{(k)})$$

$$= \overset{\text{约去}x^{(k)}}{} f(x^{(k)}) - \lambda g_k^T\cdot H_k^{-1}\cdot g_k + \underbrace{\frac{1}{2}(-\lambda H_k^{-1}\cdot g_k)^{\mathrm{T}}\cdot H_k\cdot(-\lambda H_k^{-1}\cdot g_k)}_{\text{因为}H\text{对称，}=\frac{1}{2}\lambda^2\cdot g_k^T\cdot H_k^{-1}\cdot g_k}$$

$$= f(x^{(k)}) - (\lambda - \frac{1}{2}\lambda^2)\cdot g_k^T\cdot H_k^{-1}\cdot g_k$$

所以 $f(x^{(k)}) - \mathrm{f}(\mathrm{x}^{(k+1)}) = (\lambda - \frac{1}{2}\lambda^2)\cdot g_k^T\cdot H_k^{-1}\cdot g_k$

由于 H_k 正定，所以 H_k^{-1} 正定

因此 $g_k^T\cdot H_k^{-1}\cdot g_k > 0, \text{在} 0 < \lambda < 2\text{时}, (\lambda - \frac{1}{2}\lambda^2)\cdot g_k^T\cdot H_k^{-1}\cdot g_k > 0$

$$f(x^{(k)}) - \mathrm{f}(\mathrm{x}^{(k+1)}) > 0$$

所以每次迭代后 $\mathrm{f}(\mathrm{x}^{(k+1)})$ 都是减小的，因此牛顿搜索方向是正确的搜索方向。

[16] 一维搜索(line search)的内容在第三章介绍

4.2 BFGS

Quasi-Newton 的方法有很多种，比如 DFP、BFGS、SR1 等，由于 BFGS 是被证明最有效最被广泛使用的一种算法，因此我们也着重挑 BFGS 来讲。

BFGS 的全称是 Broyden–Fletcher–Goldfarb–Shanno 算法，是由这四大金刚独立发明的，最早的 Quasi-Newton 来源于 DFP 算法的开创性贡献，BFGS 算法被认为最有要创造性的算法，非常值得研究。

4.2.1 推导

BFGS 的思想核心是这样的：我们需要找到一个近似的 Hessian 矩阵，这个近似矩阵由每次迭代更新而来：

$$B_{k+1} = B_k + \Delta B_k$$

而这个近似矩阵要满足 secant equation，即：

$$B_{k+1} s_k = y_k$$

我们现在假设[17]两个 n X 1 的 n 维向量 u 和 v：

$$\Delta B_k = P_k + Q_k = \alpha \vec{u} \bullet \vec{u}^T + \beta \vec{v} \bullet \vec{v}^T$$

所以代入 secant equation：

$$(B_k + \alpha \vec{u} \cdot \vec{u}^T + \beta \vec{v} \cdot \vec{v}^T) \cdot s_k = y_k$$

$$B_k \cdot s_k + \alpha \vec{u} \cdot \vec{u}^T \cdot s_k + \beta \vec{v} \cdot \vec{v}^T \cdot s_k = y_k$$

由于矩阵连乘可以随意“加括号”

$$B_k \cdot s_k + \vec{u}(\alpha \cdot \underbrace{\vec{u}^T \cdot s_k}_{(1\times n)*(n\times 1)=1\times 1}) + \vec{v} \cdot (\beta \cdot \vec{v}^T \cdot s_k) = y_k$$

$\alpha \cdot \vec{u}^T \cdot s_k$为一个实数，为了满足方程，设置为1，而$\beta \cdot \vec{v}^T \cdot s_k = -1$

$$B_k \cdot s_k + \vec{u}(\underbrace{\alpha \cdot \vec{u}^T \cdot s_k}_{=1}) + \vec{v} \cdot (\underbrace{\beta \cdot \vec{v}^T \cdot s_k}_{=-1}) = y_k$$

再假设$\vec{u} = \mathrm{y_k}, \vec{v} = B_k \cdot s_k$即可得：

$$B_k \cdot s_k + \underbrace{\vec{u}}_{\mathrm{y_k}} - \underbrace{\vec{v}}_{B_k \cdot s_k} = y_k$$

所以$\alpha \cdot \mathrm{y_k}^T \cdot s_k = 1 \Rightarrow \alpha = \dfrac{1}{\mathrm{y_k}^T \cdot s_k}$

$$\beta \cdot \vec{v}^T \cdot s_k = \beta \cdot (B_k \cdot s_k)^T \cdot s_k = -1 \Rightarrow \beta = -\frac{1}{(B_k \cdot s_k)^T \cdot s_k}$$

代入$\Delta B_k = \alpha \vec{u} \bullet \vec{u}^T + \beta \vec{v} \bullet \vec{v}^T = \dfrac{\mathrm{y_k} \cdot \mathrm{y_k^T}}{\mathrm{y_k}^T \cdot s_k} - \dfrac{B_k \cdot s_k \cdot s_k^T \cdot B_k}{s_k^T \cdot B_k^T \cdot s_k}$

[17]这两个向量 u 和 v 便是 BFGS 推导的神来之笔

最终迭代方程为 $B_{k+1}=B_k+\frac{\mathrm{y_k}\cdot\mathrm{y_k^T}}{\mathrm{y_k}^T\cdot s_k}-\frac{B_k\cdot s_k\cdot s_k^T\cdot B_k}{s_k^T\cdot B_k^T\cdot s_k}$

这样 Hessian 矩阵的近似矩阵 B_k 就做出来了，注意 $\mathrm{y_k}^T\cdot s_k$ 是一个实数。但是我们在迭代过程中使用的是 H^{-1}，即 Hessian 矩阵的逆矩阵。

连续应用两次 Sherman-Morrison 公式即可得到如下公式[18]，（并将 B_{k+1}^{-1} 用 D_{k+1} 来替代符号）：

$$D_{k+1}=(I-\frac{s_k y_k^T}{\mathrm{y}_k^T s_k})D_k(I-\frac{y_k s_k^T}{\mathrm{y}_k^T s_k})+\frac{s_k s_k^T}{\mathrm{y}_k^T s_k}$$

这就是 BFGS 的迭代更新 D_k 的公式，我们顺便来分析一下在 LR 中这个公式里的维数[19]：

$$D_{k+1}=(\overset{n\times n}{I}-\underset{\text{实数}}{\frac{s_k y_k^T}{\mathrm{y}_k^T s_k}})D_k(\overset{(n\times1)*(1\times n)=n\times n}{I-\underset{\text{实数}}{\frac{y_k s_k^T}{\mathrm{y}_k^T s_k}}})+\underset{\text{实数}}{\overset{n\times n}{\frac{s_k s_k^T}{\mathrm{y}_k^T s_k}}}$$

我们通常可以设置 $\rho_k=(\mathrm{y}_k^T s_k)^{-1}$ 以方便推导。

由于上面的公式有 n X n 的三个矩阵相乘，因此效率并不是很高，不过我们展开这个矩阵的连乘便可得到：

$$D_{k+1}=D_k+\frac{(\overset{\text{实数}}{s_k^T y_k}+\overbrace{y_k^T D_k y_k}^{(1\times n)*(n\times n)*(n\times 1)=\text{实数}})(\overset{n\times n}{s_k s_k^T})}{\underset{\text{实数}}{(s_k^T y_k)^2}}-\frac{\overbrace{D_k y_k s_k^T}^{(n\times n)*(n\times 1)*(1\times n)=n\times n}+\overbrace{s_k(y_k^T D_k)}^{n\times n}}{\underset{\text{实数}}{s_k^T y_k}}$$

上面这个迭代更新公式在实际编程代码中执行效率更高。

为了加深对这个更新公式的**正定性的理解**，我们再来做一下证明。

求证：如果迭代上一轮的近似矩阵 D_{k-1} 是正定矩阵，那么迭代更新后的 D_k 也是正定矩阵。

证明：

当 D_{k-1} 是正定矩阵，所以对于任意非零向量 $\vec{z}$，$\vec{z}^T\bullet D_{k-1}\bullet\vec{z}>0$，我们即是要证明 $\vec{z}^T\bullet D_k\bullet\vec{z}>0$ 也成立。

构造向量 $\vec{w}=\vec{z}-\rho_{k-1}\vec{y}_{k-1}(\vec{s}_{k-1}^T\cdot\vec{z})$，该向量即为上式中的 D_{k-1} 右边的式子右乘 $\vec{z}$

[18] 这一步有很多 trick，在附录中介绍

[19] 比如 A 矩阵是 p X q 的维度，B 矩阵是 q X r 的维度，则 A·B=p X r 的维度

$$\vec{z}^{T}\bullet D_k\bullet\vec{z}=\underbrace{z^{T}\bullet(I-\rho_{k-1}s_{k-1}y_{k-1}^{T})}_{z^{T}-\rho_{k-1}z^{T}\cdot s_{k-1}y_{k-1}^{T}=w^{T}}D_{k-1}\underbrace{(I-\rho_{k-1}y_{k-1}s_{k-1}^{T})\bullet z}_{z-\rho_{k-1}z\cdot y_{k-1}s_{k-1}^{T}=w}+z^{T}\bullet\rho_{k-1}s_{k-1}s_{k-1}^{T}\bullet z$$ [20]

$$=w^{T}D_{k-1}w+\rho_{k-1}(s_{k-1}^{T}\bullet z)^{2}$$

ρ_{k-1} 在凸函数中均为>0 的正数，又因为前提条件 D_{k-1} 是正定矩阵，所以式子中的 $w^{T}D_{k-1}w>0$，从而 $\vec{z}^{T}\bullet D_k\bullet\vec{z}>0$，得证（QED）。[21]

补充证明：为何凸函数中 ρ_{k-1}>0?

证明：

因为 Hessian 矩阵为半正定[22]，所以函数 f 一定为凸函数(convex function)。（类似于一维函数中 $d^2y/dx^2>0$ 可以判定函数为凸函数(convex function)）

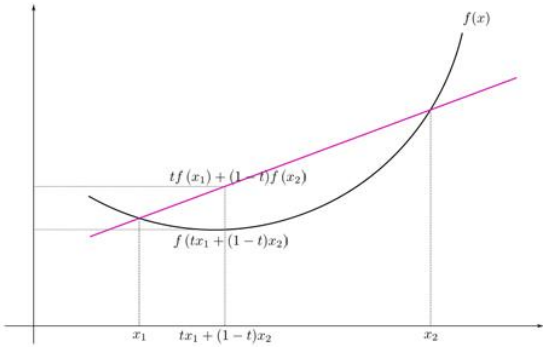


图 4.1 convex function

$$\rho_k=\frac{1}{(\mathrm{y}_k^{T}\mathrm{s}_k)}$$

$$=((\underbrace{g_{k+1}-g_k}_{1\times n})^{T}\cdot(\underbrace{\theta^{(k+1)}-\theta^{(k)}}_{n\times 1}))^{-1}$$

这里假设凸函数如上图(一维函数)，

$$\theta^{(k+1)}=\mathrm{x}_2,\quad \theta^{(k)}=\mathrm{x}_1$$

$$g_{k+1}=\mathrm{f}'(x_2),g_k=\mathrm{f}'(x_1)$$

$$\rho_k=((\overbrace{(\underbrace{\mathrm{f}'(x_2)-\mathrm{f}'(x_1)}_{>0})\bullet(\underbrace{\mathrm{x}_2-\mathrm{x}_1}_{>0})}^{\because d^2y/dx^2>0,\therefore>0}))^{-1}$$

$$\rho_k>0$$

若交换x_1和x_2的指代，反之亦然:ρ_k同样大于0

由于多维（多元）函数里的每一维的乘积项都>0，所以得证。

---

[20] 矩阵乘法分配率：A(B+C)=A·B+ A·C
矩阵相加的转置：$(A+B)^{T}=A^{T}+B^{T}$
矩阵连乘的转置：$(A\cdot B\cdot C)^{T}=(A\cdot(B\cdot C))^{T}=(B\cdot C)^{T}\cdot A^{T}=C^{T}\cdot B^{T}\cdot A^{T}$

[21] $y_{k-1}^{T}\bullet z=0$ 时上式右边项=0，只有在 w=z≠0 才可达到。因为
$\vec{w}=\vec{z}-\rho_{k-1}\vec{s}_{k-1}(\vec{y}_{k-1}^{T}\cdot\vec{z}),若\vec{w}=\vec{z},则\rho_{k-1}\vec{s}_{k-1}(\vec{y}_{k-1}^{T}\cdot\vec{z})=0,显然\vec{y}_{k-1}^{T}\cdot\vec{z}=0$

[22] 正定矩阵一定是半正定矩阵，但半正定矩阵不是正定矩阵

从这个证明中我们可以得到如下结论：只要构造初始矩阵为正定矩阵，后续的迭代过程中每次更新的近似矩阵都是正定矩阵，从而保证我们的下降搜索方向总是正确的方向。一个好的初始矩阵的**建议是使用单位矩阵**。特别在 LR 中，单位矩阵=n X n 的维度，其中 n 是待估计参数个数。

4.2.2 实现

整个算法的实现时采用一边迭代寻找 f(x)极小值，一边“**顺便**”更新 D_k 以校正方向的过程。

算法 4.1 BFGS 算法程序实现

输入：目标函数 f(x)，梯度函数 $g(x)=\nabla f(x)$，精度要求 ϵ

输出：f(x)的极小点 x^*

(1) 选定初始点 x_0（可随机选），取 D_0=I（n X n 单位矩阵），k=0

(2) 确定搜索方向 $p_k=-D_k \cdot g_k$

(3) 利用一维搜索(line search)确定步长 α_k，令 $s_k=\alpha_k \cdot p_k$；$x_{k+1} := x_k+s_k$

(4) 若$||g_{k+1}||<\epsilon$，则算法结束

(5) 计算 $y_k = g_{k+1}-g_k$

(6) 计算 $D_{k+1} = D_k + \frac{(s_k^T y_k + y_k^T D_k y_k)(s_k s_k^T)}{(s_k^T y_k)^2} - \frac{D_k y_k s_k^T + s_k (y_k^T D_k)}{s_k^T y_k}$

其中 $(s_k s_k^T)$ 为 n×n，$(s_k^T y_k)^2$ 为实数，$D_k y_k s_k^T$ 为 n×n，$s_k (y_k^T D_k)$ 为 n×n，$s_k^T y_k$ 为实数

(7) 令 k := k+1，转(2)

我以文本分类器为例，我的文本分类代码中，以每篇文章的每个 word 的 TF-IDF 值作为特征，而 $\vec{\theta}$ 向量是指每个 word 前的权重，则 θ_1 对应 wordid=1 的词的权重，python 代码如下（使用了 numpy 作为矩阵计算）：

```
def lr_bfgs(tfidfmatrix, theta, predict_tfidf):
        epsilon = 0.001
        # n * n identity matrix as Hessian's reverse
    D_matrix = np.identity(len(theta))
        identity_mat =    np.identity(len(theta))
        #    n * 1 gradient matrix
    g_matrix = calculate_gradient(tfidfmatrix, theta)
    for iter_time in xrange(16):
            print "iter_time : %s"%(iter_time)
            d_k = -np.dot(D_matrix, g_matrix) #d_k = n * 1 matrix

            #line search method to determine step size : alpha
            #theta_k, s_k = line_search_golden(tfidfmatrix, theta, d_k)
            theta_k = theta_plus_func(theta, 1.0, d_k)
            s_k = d_k
            g_matrix_k = calculate_gradient(tfidfmatrix, theta_k)
            y_k = g_matrix_k - g_matrix # n * 1 gradient matrix
```

```
        #(1)first formula    @see:
https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm
        #y_kt__s_k = float(np.dot(np.transpose(y_k), s_k)) # real number
        #print "y_kt__s_k : %s"%(y_kt__s_k)
        #if y_kt__s_k < 0.00001:
        #      break
        #using sherman morrison : first formula
        #D_matrix_k = np.dot( np.dot(identity_mat - np.dot(s_k, np.transpose(y_k))/y_kt__s_k, D_matrix),
        #      identity_mat - np.dot(y_k, np.transpose(s_k))/y_kt__s_k) + np.dot(s_k,
np.transpose(s_k))/y_kt__s_k

        #expension from first formula
        #(2)second formula    @see:
https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm
        s_kt = np.transpose(s_k)
        y_kt = np.transpose(y_k)
        s_kt_y_k = float(np.dot(s_kt, y_k))
        if s_kt_y_k < 0.00001:
            break
        D_matrix_k = D_matrix + (s_kt_y_k + float(np.dot( np.dot(y_kt, D_matrix), y_k))) * (np.dot(s_k,
s_kt))/(s_kt_y_k) ** 2 ¥
                    - (np.dot(np.dot(D_matrix, y_k),s_kt) + np.dot(s_k, np.dot(y_kt,D_matrix))) / s_kt_y_k

        D_matrix = D_matrix_k
        theta = theta_k
        g_matrix = g_matrix_k
```

在这个迭代代码中，调用的几个子函数如下：

```
def calculate_gradient(tfidfmatrix, theta):
    g_matrix_dict = defaultdict(float)
    #m * n loop
    for m,(tfidf,y) in enumerate(tfidfmatrix):
        inner = inner_product(tfidf, theta)
        for wordid,x in tfidf.items():
            g_matrix_dict[wordid] += x *(y - sigmoid(tfidf, theta, inner))
    #NOTE : minus sign
    g_matrix = np.array([-g_matrix_dict[wordid] for wordid in sorted(g_matrix_dict.keys())])
    g_matrix.shape = (1, g_matrix.size)
    return np.transpose(g_matrix) # n * 1 matrix

def theta_plus_func(theta,alpha,d_k):
    new_theta = {}
    for wordid,score in theta.iteritems():
```

```
            new_theta[wordid] = score + alpha * d_k[wordid]
        return new_theta
```

其中的 sigmoid 函数和向量内积代码如下，为了加快执行效率，sigmoid 函数多了一个内积参数，用于如果已经计算过向量内积不用重复计算：

```
def inner_product(x,y):
    sum = 0.0
    for _x, _score in x.iteritems():
        sum += _score * y[_x]
    return sum


def sigmoid(x,y,inner=None):
    if inner is None:
        inner = inner_product(x,y)
    #don't use bias theta 0, directly use inner product
    result = 1.0 / (1+ math.e ** (-(inner)))
    return result
```

结论：BFGS 适用于参数 $\vec{\theta}$ 向量的个数（维数）不是很多的情况下，可以使用 BFGS 算法快速地进行迭代训练。

4.3 L-BFGS

BFGS 算法一个问题就是当参数 $\vec{\theta}$ 向量的个数（维数）过多的时候，存储 D_k 矩阵需要 n*n 的内存空间，这在某些文本分类的情况里比较明显，语料中的词典单词数通常都达到几万级别，比如说考虑 n=10 万的情况，且用 double（8 个字节）来存储 D_k 矩阵，耗费内存如下：

$$\frac{n\text{阶矩阵字节数}}{1\text{Gb字节数}} = \frac{10^5 \times 10^5 \times 8}{2^{10} \times 2^{10} \times 2^{10}} = 74.5Gb$$

这在某些情况下是不可接受的，而且如此之大的矩阵相乘也非常耗时，导致 BFGS 算法虽然迭代次数明显比梯度下降少，但每次迭代时为了更新 D_k 花时间太长。

这就导致了 L-BFGS(Limited Memory BFGS)算法的诞生。

4.3.1 推导和实现

只需要一点毅力，便可从 BFGS 算法推导到 L-BFGS 算法。

我们从上一节的公式(FIXME)开始：

$$D_{k+1} = (\underbrace{I - \frac{s_k y_k^T}{y_k^T s_k}}_{V_k^T}) D_k (\underbrace{I - \frac{y_k s_k^T}{y_k^T s_k}}_{V_k}) + \frac{s_k s_k^T}{y_k^T s_k}$$

$$= V_k^T \cdot D_k \cdot V_k + \rho_k s_k s_k^T$$

设 D_0=I，我们不断用递推公式从 D_0、D_1、…开始展开，探寻规律

$$D_1 = \mathrm{V}_0^T \cdot D_0 \cdot \mathrm{V}_0 + \rho_0 s_0 s_0^T$$

$$D_2 = \mathrm{V}_1^T \cdot D_1 \cdot \mathrm{V}_1 + \rho_1 s_1 s_1^T$$

$$= \mathrm{V}_1^T \cdot (\mathrm{V}_0^T \cdot D_0 \cdot \mathrm{V}_0 + \rho_0 s_0 s_0^T) \cdot \mathrm{V}_1 + \rho_1 s_1 s_1^T$$

$$= \mathrm{V}_1^T \cdot \mathrm{V}_0^T \cdot D_0 \cdot \mathrm{V}_0 \cdot \mathrm{V}_1 + \mathrm{V}_1^T \cdot \rho_0 s_0 s_0^T \cdot \mathrm{V}_1 + \rho_1 s_1 s_1^T$$

$$D_3 = \mathrm{V}_2^T \cdot \mathrm{V}_1^T \cdot \mathrm{V}_0^T \cdot D_0 \cdot \mathrm{V}_0 \cdot \mathrm{V}_1 \cdot \mathrm{V}_2 + \mathrm{V}_2^T \cdot \mathrm{V}_1^T \cdot \rho_0 s_0 s_0^T \cdot \mathrm{V}_1 \cdot \mathrm{V}_2 + \mathrm{V}_2^T \cdot \rho_1 s_1 s_1^T \cdot \mathrm{V}_2 + \rho_2 s_2 s_2^T$$

很快发现规律

$$\mathrm{D}_{k+1} = (\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T \cdot \mathrm{V}_{k-2}^T \cdot ... \cdot \mathrm{V}_0^T) \cdot D_0 \cdot (\mathrm{V}_0 \cdot \mathrm{V}_1 \cdot \mathrm{V}_2 \cdot ... \cdot \mathrm{V}_k)$$

有规律

$$\begin{cases} +(\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T \cdot \mathrm{V}_{k-2}^T \cdot \mathrm{V}_{k-3}^T ... \cdot \underbrace{\mathrm{V}_1^T}_{\text{注意下标关系}}) \cdot (\rho_0 s_0 s_0^T) \cdot (\mathrm{V}_1 \cdot \mathrm{V}_2 \cdot \mathrm{V}_3 \cdot \mathrm{V}_4 \cdot ... \cdot \mathrm{V}_k) \\ +(\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T \cdot \mathrm{V}_{k-2}^T \cdot ... \cdot \mathrm{V}_2^T) \cdot (\rho_1 s_1 s_1^T) \cdot (\mathrm{V}_2 \cdot \mathrm{V}_3 \cdot \mathrm{V}_4 \cdot ... \cdot \mathrm{V}_k) \\ +... \\ +(\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T) \cdot (\rho_{k-2} s_{k-2} s_{k-2}^T) \cdot (\mathrm{V}_{k-1} \cdot \mathrm{V}_k) \\ +(\mathrm{V}_k^T) \cdot (\rho_{k-1} s_{k-1} s_{k-1}^T) \cdot (\mathrm{V}_k) \end{cases}$$

$$+ \rho_k s_k s_k^T$$

由此可见，计算 D_{k+1} 只需要用到 $\{s_i, y_i\}_{i=0}^{k}$，即需要 k 组 $\{s_i, y_i\}$ pair，因此通过分析，有限内存时 L-BFGS 并不直接在内存存储 n * n 的 D_{k+1} 矩阵，只需要一些 $\{s_i, y_i\}$ pair，保留哪几组 $\{s_i, y_i\}$ 呢？一个自然的想法当然是丢弃最早生成的 $\{s_i, y_i\}$，**只保留最近**迭代时的 M 组 $\{s_i, y_i\}$，即 $\{s_i, y_i\}_{i=k-m+1}^{k}$。这样就可以计算出近似的 D_{k+1} 了，而如果刚开始迭代少于 M 词时，则 L-BFGS 与 BFGS 完全相同，所以取 $\hat{M} = \min(M, k)$，作为保存 pair 数。

因此改造原先的 D_{k+1} 公式，如下：

$$\mathrm{D}_{k+1} = (\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T \cdot \mathrm{V}_{k-2}^T \cdot ... \cdot \mathrm{V}_{k-\hat{M}+1}^T) \cdot D_0^{(k)} \cdot (\mathrm{V}_{k-\hat{M}+1} \cdot \mathrm{V}_{k-\hat{M}+2} \cdot \mathrm{V}_{k-\hat{M}+3} \cdot ... \cdot \mathrm{V}_k)$$

最近 M 组

$$\begin{cases} +(\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T \cdot \mathrm{V}_{k-2}^T \cdot \mathrm{V}_{k-3}^T ... \cdot \underbrace{\mathrm{V}_{k-\hat{M}+2}^T}_{\text{注意下标关系}}) \cdot (\rho_{k-\hat{M}+1} s_{k-\hat{M}+1} s_{k-\hat{M}+1}^T) \cdot (\mathrm{V}_{k-\hat{M}+2} \cdot \mathrm{V}_{k-\hat{M}+3} \cdot \mathrm{V}_{k-\hat{M}+4} \cdot \mathrm{V}_{k-\hat{M}+5} \cdot ... \cdot \mathrm{V}_k) \\ +(\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T \cdot \mathrm{V}_{k-2}^T \cdot ... \cdot \mathrm{V}_{k-\hat{M}+3}^T) \cdot (\rho_{k-\hat{M}+2} s_{k-\hat{M}+2} s_{k-\hat{M}+2}^T) \cdot (\mathrm{V}_{k-\hat{M}+3} \cdot \mathrm{V}_{k-\hat{M}+4} \cdot \mathrm{V}_{k-\hat{M}+5} \cdot ... \cdot \mathrm{V}_k) \\ +... \\ +(\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T \cdot \mathrm{V}_{k-2}^T) \cdot (\rho_{k-3} s_{k-3} s_{k-3}^T) \cdot (\mathrm{V}_{k-2} \cdot \mathrm{V}_{k-1} \cdot \mathrm{V}_k) \\ +(\mathrm{V}_k^T \cdot \mathrm{V}_{k-1}^T) \cdot (\rho_{k-2} s_{k-2} s_{k-2}^T) \cdot (\mathrm{V}_{k-1} \cdot \mathrm{V}_k) \\ +(\mathrm{V}_k^T) \cdot (\rho_{k-1} s_{k-1} s_{k-1}^T) \cdot (\mathrm{V}_k) \end{cases}$$

$$+ \rho_k s_k s_k^T$$

如何写程序生成这个 D_{k+1} 呢？下面来分析用**精妙绝伦**的二次循环法构造这个 D_{k+1}：

算法 4.2 L-BFGS 确定 p_k 搜索方向的二次循环算法(two loop method)[23]

返回值：搜索方向 $D_{k+1}\cdot g_{k+1}$

//后向循环

①Initialize $q=g_{k+1}$

For i=0 to M-1 Do

{

② $a_i = \rho_{k-i} s_{k-i}^T q$ //a_i保存下来，前向循环要用

③ $q = q - \alpha_i y_{k-i}$

}

//前向循环

④ $P = H_{k-m} q = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} \cdot q$

For i=M-1 to 0 Do

{

⑤ $\beta = \rho_{k-i} y_{k-i}^T P$ //β是实数

⑥ $P = P + s_{k-i}(a_i - \beta)$

}

⑦ return P

为了分析方便，我将上述程序的每个语句都用圆圈+数字方式进行了标号

分析二次循环：

(1) 第一次循环分析

我们再次搬出前面的那个大式子(不要怕麻烦)，交换[24]乘积中间的因子的项目为 $s_i \rho_i s_i^T$，并且等式左右两边同时右乘 g_{k+1}。

在正式分析之前，请**注意**，a_i 是一个实数，q_i 是一个 n X 1 的向量。$D_0^{(k)}$设置为 $\frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} \cdot I$ (n X n 维)比较合适，而根据 n X n 的对角矩阵(diagonal matrix)和 n X 1 的矩阵 $(V_{k-\hat{M}+1} \cdot V_{k-\hat{M}+2} \cdot V_{k-\hat{M}+3} \cdot \ldots \cdot V_k) \cdot g_{k+1}$的相乘法则，等价于直接使用 $\frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}$ 这个数字乘以后面的 n X 1 矩阵 $(V_{k-\hat{M}+1} \cdot V_{k-\hat{M}+2} \cdot V_{k-\hat{M}+3} \cdot \ldots \cdot V_k) \cdot g_{k+1}$更节省内存和快速。（这也是 L-BFGS 之所以比 BFGS 节省内存的**关键所在**：n X n 矩阵存储转变成了 n X 1 的向量）

[23] 整个算法 4.2 就是为构造 $D_{k+1} \cdot g_{k+1}$ 公式而生

[24] ρ_i 是一个小正实数，因此这是简单的乘法交换律

$$D_{k+1}g_{k+1}=(V_k^T\cdot V_{k-1}^T\cdot V_{k-2}^T\cdot\ldots\cdot V_{k-\hat{M}+1}^T)\cdot \underbrace{\frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}}\cdot I}_{D_0^{(k)}}\cdot\overbrace{(V_{k-\hat{M}+1}\cdot V_{k-\hat{M}+2}\cdot V_{k-\hat{M}+3}\cdot\ldots\cdot V_k)\cdot g_{k+1}}^{\text{finish 1st loop.}=q_{\hat{M}-1}}$$

最近M组

$$+(V_k^T\cdot V_{k-1}^T\cdot V_{k-2}^T\cdot V_{k-3}^T\ldots\cdot V_{k-\hat{M}+2}^T)\cdot(s_{k-\hat{M}+1}\underbrace{\rho_{k-\hat{M}+1}s_{k-\hat{M}+1}^T)\cdot(V_{k-\hat{M}+2}\cdot V_{k-\hat{M}+3}\cdot V_{k-\hat{M}+4}\cdot V_{k-\hat{M}+5}\cdot\ldots\cdot V_k)\cdot g_{k+1}}_{\text{finish 1st loop.}=a_{\hat{M}-1}}$$

$$+(V_k^T\cdot V_{k-1}^T\cdot V_{k-2}^T\cdot\ldots\cdot V_{k-\hat{M}+3}^T)\cdot(s_{k-\hat{M}+2}\underbrace{\rho_{k-\hat{M}+2}s_{k-\hat{M}+2}^T)\cdot\overbrace{(V_{k-\hat{M}+3}\cdot V_{k-\hat{M}+4}\cdot V_{k-\hat{M}+5}\cdot\ldots\cdot V_k)\cdot g_{k+1}}^{=q_{\hat{M}-3}}}_{\text{注意}\rho_{k-\hat{M}+2}s_{k-\hat{M}+2}^T\text{下标是}k-(\hat{M}-2)\text{，所以}=a_{\hat{M}-2}}$$

$$+\ldots$$

$$+(V_k^T\cdot V_{k-1}^T\cdot V_{k-2}^T)\cdot(s_{k-3}\rho_{k-3}s_{k-3}^T)\cdot(V_{k-2}\cdot V_{k-1}\cdot V_k)\cdot g_{k+1}$$

$$+(V_k^T\cdot V_{k-1}^T)\cdot(s_{k-2}\rho_{k-2}s_{k-2}^T)\cdot(V_{k-1}\cdot V_k)\cdot g_{k+1}$$

$$+(V_k^T)\cdot(s_{k-1}\underbrace{\rho_{k-1}s_{k-1}^T)\cdot\overbrace{(V_k)\cdot g_{k+1}}^{q_0}}_{a_1}$$

$$+s_k\underbrace{\rho_k s_k^T\cdot g_{k+1}}_{a_0}$$

代码里 q 和 α 的含义从上面公式中标红的部分一看便知，我们先来分析一下 q

$q_0=g_{k+1}$

$$q_i=(V_{k-i}\cdot V_{k-i+1}\cdot V_{k-i+2}\cdot V_{k-i+3}\cdot\ldots\cdot V_k)\cdot g_{k+1}$$

$$q_{i-1}=(V_{k-i+1}\cdot V_{k-i+2}\cdot V_{k-i+3}\cdot\ldots\cdot V_k)\cdot g_{k+1}$$

所以，q_i 的递推推导如下：

$$q_i=V_{k-i}\cdot q_{i-1}=(I-\frac{y_{k-i}s_{k-i}^T}{y_{k-i}^T s_{k-i}})\cdot q_{i-1}$$

$$=q_{i-1}-\rho_{k-i}y_{k-i}s_{k-i}^T\cdot q_{i-1}$$

$$=q_{i-1}-y_{k-i}\cdot a_i$$

1st loop 后向循环执行时：

i=0 $a_0=\rho_k s_k^T g_{k+1}$ ； $q=g_{k+1}-\rho_k s_k^T g_{k+1}y_k=V_k\cdot g_{k+1}$ //注释：[25]

…

i=M-1

$$\begin{cases}a_{\hat{M}-1}=(\rho_{k-\hat{M}+1}s_{k-\hat{M}+1}s_{k-\hat{M}+1}^T)\cdot(V_{k-\hat{M}+2}\cdot V_{k-\hat{M}+3}\cdot V_{k-\hat{M}+4}\cdot V_{k-\hat{M}+5}\cdot\ldots\cdot V_k)\cdot g_{k+1}\\ q=(V_{k-\hat{M}+1}\cdot V_{k-\hat{M}+2}\cdot V_{k-\hat{M}+3}\cdot\ldots\cdot V_k)\cdot g_{k+1}\end{cases}$$

分析结论：所以，前向循环是**从下往上构造公式的右半部份**的，启动时 a 和 q 前后相差一项，共经历 M 次循环（整个公式有 M+1 个加法项），q 每次更新，而 a 是个 double 数组(数组长度 M)，保存每次的 a_i。

[25] 这里如果你在纸上推导一下，有些人就会发出疑问乘法顺序不一致，其实原因很简单，记住 a_0 和 ρ_k 都是实数即可满足乘法交换律

(2) 第二次循环分析

现在执行第④句，可以看到第④句执行之后:

$$\mathrm{P} = D_0^{(k)} \cdot (\mathrm{V}_{k-\hat{M}+1} \cdot \mathrm{V}_{k-\hat{M}+2} \cdot \mathrm{V}_{k-\hat{M}+3} \cdot \ldots \cdot \mathrm{V}_k) \cdot g_{k+1}$$

我们先用递推公式： $\mathrm{P} := V_{k-i}^T \cdot P + S_{k-i} a_i$ 来进行迭代

紧接着循环开始

2nd loop 前向循环执行时：

i=M-1 ： $P = \mathrm{V}_{k-\hat{M}+1}^T \cdot D_0^{(k)} \cdot (\mathrm{V}_{k-\hat{M}+1} \cdot \mathrm{V}_{k-\hat{M}+2} \cdot \mathrm{V}_{k-\hat{M}+3} \cdot \ldots \cdot \mathrm{V}_k) \cdot g_{k+1} + s_{k-\hat{M}+1} a_{\hat{M}-1}$

自动相乘

i=M-2 ： $P = \mathrm{V}_{k-\hat{M}+2}^T \cdot P + s_{k-\hat{M}+2} a_{\hat{M}-2}$

自动相乘

i=M-3 ： $P = \mathrm{V}_{k-\hat{M}+3}^T \cdot P + s_{k-\hat{M}+3} a_{\hat{M}-3}$

…

i=0 ： $P = \mathrm{V}_k^T \cdot P + s_k a_0$ //这样 $s_k a_0$ 就构造出了最后一项

通过这个递推公式的循环分析可以知道，“隐式的自动相乘”方法在这个循环里起着重要的作用。注意到这个递推公式(FIXME)中由于含有 $V_{k-i}^T \cdot P$ （n X n 维矩阵 * n X 1 维矩阵）矩阵相乘较慢，因此可以优化，算法的第⑤和第⑥句其实和递推公式(FIXME)是等价的：

$$\mathrm{P} := V_{k-i}^T \cdot P + s_{k-i} a_i$$

$$\text{其中} V_{k-i}^T = I - \frac{s_{k-i} y_{k-i}^T}{y_{k-i}^T s_{k-i}}$$

$$P := P - \frac{s_{k-i} y_{k-i}^T}{y_{k-i}^T s_{k-i}} P + s_{k-i} a_i$$

$$= P - \rho_{k-i} s_{k-i} y_{k-i}^T P + s_{k-i} a_i$$

$$= P + s_{k-i} (a_i - \underbrace{\rho_{k-i} y_{k-i}^T P}_{\beta})$$

所以算法 4.2 中用的是优化后的递推公式，整个前向循环是从上到下构造方向 $\mathrm{D}_{k+1} g_{k+1}$。

有了确定方向的算法 4.2，我们可以写出整个 L-BFGS 的算法主程序了：

算法 4.3 L-BFGS 算法

输入：起始点 x_0，integer history size M > 0， k=1

输出：目标函数 f(x)的极小值点 x^*

while no converge do
{
    计算 x_k 点处的梯度 g_k
    用算法 4.2 计算方向-p_k
    使用一维搜索(line search)计算 α_k
    用公式 $x_{k+1}=x_k+\alpha_k \cdot p_k$
    if k > m then
    {
        从内存中丢弃(discard)最早的 vector pair s_{k-m}, y_{k-m}
    }
    update $s_k=x_{k+1}-x_k$， $y_k=g_{k+1}-g_k$， k = k+1;
}

附录

推导：Sherman-Morrison 公式 → D_k

Sherman Morrison 公式：

$$\left(\mathrm{A}+\frac{uu^T}{t}\right)^{-1}=A^{-1}-\frac{A^{-1}uu^TA^{-1}}{t+u^TA^{-1}u}$$

$$\left(\mathrm{H}+\frac{yy^T}{y^T\mathrm{s}}-\frac{Hss^T\mathrm{H}}{s^THs}\right)^{-1}$$

$$=\left(\mathrm{H}+\frac{yy^T}{y^T\mathrm{s}}\right)^{-1}+\left(\mathrm{H}+\frac{yy^T}{y^T\mathrm{s}}\right)^{-1}\frac{Hss^TH}{s^TH^Ts-s^TH\left(\mathrm{H}+\frac{yy^T}{y^T\mathrm{s}}\right)^{-1}Hs}\left(\mathrm{H}+\frac{yy^T}{y^T\mathrm{s}}\right)^{-1}$$

$$=(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})+(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})\frac{Hss^TH}{s^THs-s^TH(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})Hs}(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})$$

$$=(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})+(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})\frac{Hss^TH}{\frac{s^Tyy^Ts}{y^T\mathrm{s}+y^TH^{-1}y}}(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})$$

$$=(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})+\frac{H^{-1}Hss^THH^{-1}}{\frac{s^Tyy^Ts}{y^T\mathrm{s}+y^TH^{-1}y}}-\frac{H^{-1}Hss^TH}{\frac{s^Tyy^Ts}{y^T\mathrm{s}+y^TH^{-1}y}}H^{-1}\frac{yy^T}{y^T\mathrm{s}+y^TH^{-1}y}H^{-1}$$

$$-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y}\frac{Hss^TH}{\frac{s^Tyy^Ts}{y^T\mathrm{s}+y^TH^{-1}y}}H^{-1}$$

$$+H^{-1}\frac{yy^T}{y^T\mathrm{s}+y^TH^{-1}y}H^{-1}\frac{Hss^TH}{\frac{s^Tyy^Ts}{y^T\mathrm{s}+y^TH^{-1}y}}H^{-1}\frac{yy^T}{y^T\mathrm{s}+y^TH^{-1}y}H^{-1}$$

$$=(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})+\frac{ss^T(y^T\mathrm{s}+y^TH^{-1}y)}{s^Tyy^Ts}-\frac{ss^Tyy^TH^{-1}}{s^Tyy^Ts}-\frac{H^{-1}yy^Tss^T}{s^Tyy^Ts}$$

$$+\frac{H^{-1}yy^Tss^Tyy^TH^{-1}}{(y^T\mathrm{s}+y^TH^{-1}y)s^Tyy^Ts}$$

$$=(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})+\frac{ss^T(y^T\mathrm{s}+y^TH^{-1}y)}{(s^Ty)^2}-\frac{s(s^Ty)y^TH^{-1}}{(s^Ty)^2}-\frac{H^{-1}y(y^Ts)s^T}{(s^Ty)^2}$$

$$+\frac{H^{-1}y(y^Tss^Ty)y^TH^{-1}}{(y^T\mathrm{s}+y^TH^{-1}y)s^Tyy^Ts}$$

$$=(H^{-1}-\frac{H^{-1}yy^TH^{-1}}{y^T\mathrm{s}+y^TH^{-1}y})+\frac{ss^T(y^T\mathrm{s}+y^TH^{-1}y)}{(s^Ty)^2}-\frac{sy^TH^{-1}}{s^Ty}-\frac{H^{-1}ys^T}{s^Ty}+\frac{H^{-1}yy^TH^{-1}}{(y^T\mathrm{s}+y^TH^{-1}y)}$$

$$=H^{-1}+\frac{ss^T(y^T\mathrm{s}+y^TH^{-1}y)}{(s^Ty)^2}-\frac{sy^TH^{-1}}{s^Ty}-\frac{H^{-1}ys^T}{s^Ty}$$

$$=H^{-1}+\frac{ss^Ty^T\mathrm{s}}{(s^Ty)^2}+\frac{ss^Ty^TH^{-1}y}{(s^Ty)^2}-\frac{sy^TH^{-1}}{s^Ty}-\frac{H^{-1}ys^T}{s^Ty}$$

$$= H^{-1}\left(I - \frac{ys^T}{s^T y}\right) - \frac{sy^T H^{-1}}{s^T y}\left(I - \frac{ys^T}{s^T y}\right) + \frac{ss^T}{s^T y}$$
$$= \left(I - \frac{sy^T}{s^T y}\right) H^{-1}\left(I - \frac{ys^T}{s^T y}\right) + \frac{ss^T}{s^T y}$$

参考文献（Reference）

http://www.cnblogs.com/jeromeblog/p/3801025.html?utm_source=tuicool

http://blog.csdn.net/itplus/article/details/21896619

http://wenku.baidu.com/view/1b3a70280066f5335a8121ec.html