CRF及求解算法L-BFGS,Line Search原理及代码分析

王鹏（qqlantian@126.com）
Last updated on 2017-2-17

如需获取最新文档请访问https://github.com/AlexPengW/resource/tree/master/pdf

Abstract

本文首先介绍**CRF**[1]模型及用来提高CRF求解和预测效率的**Forward-Backward** 算法，然后介绍经典的无约束优化算法**Line Search**[2]，**DFP**[3]，**BFGS**，**L-BFGS**[4], **IIS**[5]，给出一些必要性的证明，最后结合CRF++代码讲解CRF和L-BFGS的代码实现。

CRF模型

CRF模型概述

> CRFs are a type of discriminative undirected probabilistic graphical model. It is used to encode known relationships between observations and construct consistent interpretations. It is often used for labeling or parsing of sequential data, such as natural language text or biological sequences and in computer vision. —— Wikipedia

本文主要描述Linear-chain CRF，

记$P(y|x,\lambda)$为CRF模型定义的条件概率， 即给定观测序列x时，任意标注序列y的条件概率,则$P(y|x,\lambda)$定义如下：

$$\begin{cases} P\left(y|x,\lambda\right) & = \frac{1}{Z(x)} exp\left(\sum_{i,k}\lambda_k t_k\left(y_{i-1},y_i,x,i\right)+\sum_{i,l}\mu_l s_l\left(y_i,x,i\right)\right) \\ Z\left(x\right) & = \sum_y exp\left(\sum_{i,k}\lambda_k t_k\left(y_{i-1},y_i,x,i\right)+\sum_{i,l}\mu_l s_l\left(y_i,x,i\right)\right) \end{cases} \quad (1)$$

其中$s_l(y_i,x,i)$为状态特征函数，$t_k(y_{i-1},y_i,x,i)$为转移特征函数。特征函数由用户定义，CRF++代码里面用Unigram模板提取状态特征，用Bigram模板提取转移特征，对应的特征函数返回0或1。μ_l和λ_k分别是对应的特征权值，即待求解的模型参数。

上面的概率公式具体来历可以参考统计学习方法[6]书中的最大熵模型，或者这篇博客最大熵模型[7]，这里不再叙述。

为了简化公式，记$s(y_i,x,i)=s(y_{i-1},y_i,x,i)$, $f(y_{i-1},y_i,x,i)$或者是个状态函数$s(y_{i-1},y_i,x,i)$或者是个转移函数$t(y_{i-1},y_i,x,i)$，记N为某个样本序列长度，不同样本序列长度可能不同，记L为标记序列y在任意位置可能取到的标记个数，记T为特征数量。则$P(y|x,\lambda)$也可以定义如下：

$$\begin{cases} P\left(y|x,\lambda\right) & = \frac{1}{Z(x)} exp\left(\sum_j \lambda_j F_j\left(y,x\right)\right) \\ F_j\left(y,x\right) & = \sum_{i=1}^{N} f_j\left(y_{i-1},y_i,x,i\right) \\ Z(x) & = \sum_y exp\left(\sum_j \lambda_j F_j\left(y,x\right)\right) \end{cases} \quad (2)$$

CRF模型求解

给定训练样本$\{(x^m,y^m)\},m=\{1,\ldots,M\}$, 则CRF模型的对数似然函数为：

$$\mathcal{L}(\lambda)=\sum_m logP\left(y^m|x^m,\lambda\right)=\sum_m\left(\sum_j\lambda_j F_j\left(x^m,y^m\right)-logZ\left(x^m\right)\right) \quad (3)$$

可以证明上面的函数是关于λ的凹函数，我只试了一个不优美的证明方法，所以就不列出来了，简单来说就是利用**[定理A function is convex if and only if it is convex when restricted to any line that intersects its domain，这个定理也容易证明]**将高维上的证明转化为一维上的证明，

然后证明一维函数二阶导数小于0即可。通常我们求解时都会在后面减去L_2正则化项$\frac{\sum_j \lambda_j^2}{2C}$，减去$L_2$正则化项后就是严格凹函数(strictly concave)，由严格凹函数性质知$\mathcal{L}(\lambda)$有唯一一个全局最大值，且最大值处导出为0。

将对数似然对λ_j求导得：

$$
\begin{cases}
\frac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j} = \sum_m \left(F_j\left(x^m, y^m\right) - \frac{\sum_y \left(F_j\left(y, x^m\right) exp\left(\sum_j \lambda_j F_j\left(y, x^m\right) \right) \right)}{Z(x^m)} \right) \\
\quad = \sum_m \left(F_j\left(x^m, y^m\right) - \sum_y P(y|x^m, \lambda) F_j\left(y, x^m\right) \right) \\
\quad = \sum_m \left(F_j\left(x^m, y^m\right) - E_{P(Y|x^m,\lambda)}[F_j\left(Y, x^m\right)] \right) \\
\quad = M \sum_m \left(\widetilde{P}(y^m, x^m) F_j(y^m, x^m) - \widetilde{P}(x^m) \sum_y P(y|x^m, \lambda) F_j(y, x^m) \right) \\
\quad = M \left(E_{\widetilde{P}(Y,X)}[F_j(y^m, x^m)] - E_{P(Y,X)}[F_j(Y, x^m)] \right)
\end{cases}
\tag{4}
$$

注意上面的$\widetilde{P}(Y, X)$是X，Y的联合经验概率分布， 概率之和为1，所以左边有乘以M，$P(Y, X)$是模型分布（$P(y, x) = \widetilde{P}(x) P(y|x, \lambda)$），CRF++代码是按照$\sum_m \left(F_j(x^m, y^m) - E_{P(Y|x^m,\lambda)}[F_j(Y, x^m)] \right)$来计算梯度的(准确的说CRF++会对似然函数和导数取负号，转化为最小化问题)。

上面的梯度式子中，对于每个样本要计算特征$F_j(Y, x^m)$在模型条件分布$P(Y|x^m, \lambda)$上的期望$E_{P(Y|x^m,\lambda)}[F_j\left(Y, x^m\right)]$。记$g$为梯度数组。

本文采用一种与论文不同的思路来推引出**Forward Backward**算法，写的比较绕。

暴力求解

假设用最直接的方法求梯度$\sum_m \left(F_j\left(x^m, y^m\right) - \sum_y P(y|x^m, \lambda) F_j\left(y, x^m\right) \right)$，对于后面一项，$y$有$L^N$种不同取值，对于每个取值$y$, 计算$P(y|x^m, \lambda)$，然后将计算结果累加到梯度中，即：

$$
g[j] = g[j] - F_j(y, x^m) P(y|x^m, \lambda)
$$

$F_j(y, x^m)$即第j个特征在(y, x^m)中出现的次数。计算$P(y|x^m, \lambda)$概率复杂度为$O(NT)$(这里假设提前分别统计存储每个位置上的特征，所以要遍历每个位置，共N个位置，每个位置最多可能T个特征。只考虑非规范化概率，即不考虑$Z(x)$（这项所有y都一样，可以在非规范化概率计算完后，累加即得到，不影响整体计算复杂度）；后面将概率累加到数组g，复杂度也是$O(NT)$(如前面所说每个位置特征单独存储，所以单独相加，最差情况下这个复杂度是$O(NT)$, 当然实际上，由于$P(y|x^m, \lambda)$计算时涉及计算指数和对数，实际计算时间会是$(g[])$的倍数）。所以对于每个样本总体计算复杂度是

$O(Y$集合大小$) * (O(P(y|x^m, \lambda) + O(g[])) = O\left(L^N * N^T\right)$

为了节省CPU，考虑下怎么加速，加速方法一般就是减少重复计算。很容易可以发现对于两个序列y^1, y^2, 如果他们只有最后一个位置的标记不同，那么$F_j(y,x)=\sum_{i=1}^{N} f_j(y_{i-1},y_i,x,i)$的前$N-1$部分累加和是相等的, 如果枚举$Y$时按照一定的顺序，使每次$y$只在一个位置发生变化，那$O(P(y|x^m,\lambda))$的计算时间一般情况也能得到明显提高，不过即使能降低$O(P(y|x^m,\lambda))$的计算复杂度，由于$O(g[]+=)$并未变化，$O(Y$集合大小$)$也未变化，所以整体计算复杂度未变。 **所以只要采取枚举y的计算策略，单一靠降低单个y的$O(P(y|x^m,\lambda))$的计算复杂度是不能降低整体计算复杂度的。**

Forward-Backward算法

由于枚举y的暴力计算方法无法降低总体计算复杂度，我们**可以寻求将y打包计算的方法。**比如如果能将Y打包成一些子集$\{Y^q|q=\{1,\ldots,Q\}\}$, 其中不同子集可能相交，将$\frac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j}$进行适当变换，使计算复杂度可以表示成

$$O(Q)*(O(P(Y^q|x^m,\lambda)+O(g[])) \tag{5}$$

并且使Q远小于L^N， 并且计算$O(P(Y^q|x^m,\lambda)$时不能采取之前枚举y累加的办法，如果枚举累加，那么总体复杂度还跟之前一样了，**所以各个子集的选取要方便一起打包高效计算。**

对$\frac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j}$做如下变化

$$\begin{cases} \frac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j}=\sum_m\left(F_j\left(x^m,y^m\right)-\sum_y P(y|x^m,\lambda)F_j\left(y,x^m\right)\right) \\ \quad =\sum_m\left(F_j\left(x^m,y^m\right)-\sum_{i=1}^{N}\left(\sum_y f_j\left(y_{i-1},y_i,x^m,i\right)P\left(y|x^m,\lambda\right)\right)\right) \\ \quad =\sum_m\left(F_j\left(x^m,y^m\right)-\sum_{i=1}^{N}\left(\sum_{l,l'}\left(\sum_{\{y|y_{i-1}=l\ and\ y_i=l'\}} f_j\left(y_{i-1},y_i,x^m,i\right)P\left(y|x^m,\lambda\right)\right)\right)\right) \end{cases} \tag{6}$$

如果f_j是单特征，那么上式可以进一步化简为

$$\frac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j}=\sum_m\left(F_j\left(x^m,y^m\right)-\sum_{i=1}^{N}\left(\sum_l\left(\sum_{\{y|y_i=l\}} s_j\left(y_i,x^m,i\right)P\left(y|x^m,\lambda\right)\right)\right)\right)$$

上式中的l, l'是相邻标记可能出现的各种组合,一共L^2种，右边的$\{y|y_{i-1}=l\ and\ y_i=l'\}$是为转移特征找到的一种$y$集合, 集合里面每个$y$满足$y_{i-1}=l\ and\ y_i=l'$，一共$(N-1)L^2$种$y$集合。$\{y|y_i=l\}$是我们为状态特征找到的集合，一共$NL$种, 记$Y^q$为任一找到的集合，$q\in\{1,\ldots,NL\}$为状态特征集合，$q\in\{NL+1,\ldots,(N-1)L^2+NL\}$为转移特征集合，记

$$P(Y^q|x^m,\lambda)=\sum_{\{y|y\in Y^q\}} P(y|x^m,\lambda)$$

对于状态特征

$$\frac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j}=\sum_m\left(F_j(x^m,y^m)-\sum_{\{q|q=1,\ldots,NL\}} s_j(y_i,x^m,i)P(Y^q|x^m,\lambda)\right)$$

对于转移特征

$$\frac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j} = \sum_m \left(F_j(x^m, y^m) - \sum_{\{q|q\in(NL+1,\ldots,(N-1)L^2+NL)\}} t_j(y_{i-1}, y_i, x^m, i) P(Y^q|x^m, \lambda) \right)$$

现在的问题就是寻找$P(Y^q|x^m, \lambda)$的高效算法。

记$S_{y_n=l}$为某个状态特征Y子集，$S_{y_{n-1}=l, y_n=l'}$为某个转移特征Y子集。

对于两个状态特征子集$S_{y_n=a}, S_{y_n=b}$，存在一个一一映射，对于任意$y^a \in S_{y_n=a}$，存在一个唯一的$y^b \in S_{y_n=b}$，使y^a, y^b只在第n个位置上不同，在其他位置都是一样的。考虑$S_{y_n=a}, S_{y_n=b}$的两个子集$S_{y_{n-1}=c \text{ and } y_n=a \text{ and } y_{n+1}=d}, S_{y_{n-1}=c \text{ and } y_n=b \text{ and } y_{n+1}=d}$，两个子集的元素也存在一一对应关系，使他们只在位置n不同。

记

$$\alpha_{n-1}(c) = \sum_{y\in\alpha_{n-1,c}} exp\left(\sum_j \left(\lambda_j \sum_{i=1}^{n-1} f_j\left(y_{i-1}, y_i, x, i\right) \right) \right)$$

$$\beta_{n+1}(d) = \sum_{y\in\beta_{n+1,c}} exp\left(\sum_j \left(\lambda_j \sum_{i=2}^{N-n} f_j\left(y_{i-1}, y_i, x, i\right) \right) \right)$$

$$M_n(c, a) = exp\left(\sum_j \lambda_j f_j\left(c, a, x, n\right) \right)$$

其中$\alpha_{n-1,c}$为所有长度为$n-1$,终点为c的标记序列集合，$\beta_{n+1,d}$为所有长度为$N-n$,起点为d的标记序列集合，$\alpha_{n-1}(c)$表示集合$\alpha_{n-1,c}$前向非规范化概率和，$\beta_{n+1}(d)$表示集合$\beta_{n+1,d}$的后项非规范化概率和。则

$$P(S_{y_{n-1}=c \text{ and } y_n=a \text{ and } y_{n+1}=d}|x, \lambda) = \frac{1}{Z(x)} \alpha_{n-1}(c) M_n(c, a) M_{n+1}(a, d) \beta_{n+1}(d)$$

上面的公式可以这样理解：

记$\alpha_{n,c,a}$为长度为n且最后两个标记分别为c, a的标记序列集合，则$\alpha_{n,c,a}$与$\alpha_{n-1,c}$两个集合元素个数一样，将集合$\alpha_{n-1,c}$里面每个元素尾部加个a就得到集合$\alpha_{n,c,a}$。记$\alpha_n(c, a)$表示集合$\alpha_{n,c,a}$的前向概率和，则

$$\begin{aligned}
&\alpha_n(c, a) = \sum_{y\in\alpha_{n,c,a}} exp\left(\sum_j \lambda_j \sum_{i=1}^{n} f_j\left(y_{i-1}, y_i, x, i\right) \right) \\
&= \sum_{y\in\alpha_{n,c,a}} exp\left(\sum_j \lambda_j \left(f_j\left(c, a, x, n\right) + \sum_{i=1}^{n-1} f_j\left(y_{i-1}, y_i, x, i\right) \right) \right) \\
&= exp\left(\sum_j \lambda_j f_j\left(c, a, x, n\right) \right) \sum_{y\in\alpha_{n,c,a}} exp\left(\sum_j \sum_{i=1}^{n-1} f_j\left(y_{i-1}, y_i, x, i\right) \right) \\
&= M_n(c, a) \sum_{y\in\alpha_{n,c,a}} exp\left(\sum_j \sum_{i=1}^{n-1} f_j\left(y_{i-1}, y_i, x, i\right) \right) \\
&= M_n(c, a) \alpha_{n-1}(c)
\end{aligned}$$

同理，记$\alpha_{n+1,c,a,d}$为长度为$n+1$且最后三个标记分别为c, a, d的标记序列集合,能得出

$$\alpha_{n+1}(c, a, d) = \alpha_{n-1}(c) M_n(c, a) M_{n+1}(a, d)$$

将集合$\alpha_{n+1,c,a,d}$每个元素扩充一下（每个元素尾部分别追加集合$\beta_{n+1,d}$里面的元素）便得到集合$S_{y_{n-1}=c \text{ and } y_n=a \text{ and } y_{n+1}=d}$，用类似上面推出$\alpha_n(c, a)$的方法，也可以推出（有些绕就不推了）

$$P(S_{y_{n-1}=c \text{ and } y_n=a \text{ and } y_{n+1}=d}|x, \lambda) = \frac{1}{Z(x)} \alpha_{n-1}(c) M_n(c, a) M_{n+1}(a, d) \beta_{n+1}(d)$$

同理可以推出

$$P(S_{y_{n-1}=c \text{ and } y_n=b \text{ and } y_{n+1}=d}|x, \lambda) = \frac{1}{Z(x)} \alpha_{n-1}(c) M_n(c, b) M_{n+1}(b, d) \beta_{n+1}(d)$$

$P(S_{y_{n-1}=c \text{ and } y_n=a \text{ and } y_{n+1}=d}|x, \lambda)$和$P(S_{y_{n-1}=c \text{ and } y_n=b \text{ and } y_{n+1}=d}|x, \lambda)$具有公共子结构$\alpha_{n-1}(c)$和$\beta_{n+1}(d)$，也即**存在可以复用的子结构来提高求解速度。并且由α, β定义知道子结构之间存在递推关系**：

$$\begin{cases} \alpha_n(a) = \sum_l \alpha_{n-1}(l) M_n(l,a) \\ \beta_n(a) = \sum_l \beta_{n+1}(l) M_{n+1}(a,l) \end{cases} \tag{7}$$

这样便可以用经典的动态规划算法来提高求解效率，避免子问题重复求解。

直接由$\alpha_n(a), \beta_n(a)$的定义能得出

$$P(S_{y_n=a}|x,\lambda) = \frac{1}{Z(x)} \alpha_n(a)\beta_n(a)$$

也可以将$S_{y_n=a}$分解成互不相交子集合的形式累加出来

$$P(S_{y_n=a}|x,\lambda) = \sum_{l,l'} P(S_{y_{n-1}=l \text{ and } y_n=a \text{ and } y_{n+1}=l'}|x,\lambda)$$

$$= \frac{1}{Z(x)} \sum_{l,l'} \alpha_{n-1}(l) M_n(l,a) M_{n+1}(a,l') \beta_{n+1}(l')$$

$$= \frac{1}{Z(x)} \sum_{l'} \{\beta_{n+1}(l') M_{n+1}(a,l') (\sum_l \alpha_{n-1}(l) M_n(l,a))\}$$

$$= \frac{1}{Z(x)} \sum_{l'} \{\beta_{n+1}(l') M_{n+1}(a,l') \alpha_n(a)\}$$

$$= \frac{1}{Z(x)} \alpha_n(a) \sum_{l'} \beta_{n+1}(l') M_{n+1}(a,l')$$

$$= \frac{1}{Z(x)} \alpha_n(a)\beta_n(a)$$

同理可以推出

$$P(S_{y_{n-1}=l, y_n=l'}|x,\lambda) = \frac{1}{Z(x)} \alpha_n(a) M_n(l,l') \beta_n(a)$$

这样便可以将各个Y子集$S_{y_n=l}$和$S_{y_{n-1}=l, y_n=l'}$的概率用$\alpha_n(a), \beta_n(a)$表示出来。并且由公式(7)可以用动态规划法高效求出$\alpha_n(a), \beta_n(a)$。

现在来根据公式(5)计算下**Forward-Backward**算法的总体时间复杂度。

由于多了求$\alpha_n(a), \beta_n(a)$的开销，所以总体复杂度要加上$O(\alpha_n(a) + \beta_n(a))$

第一步先计算所有$M_n(l,l')$,再计算$\alpha_n(a), \beta_n(a)$，总体计算复杂度是$O(NL^2)$

第二步计算$Z(x) = \sum_l \alpha_N(l)$为$O(L)$复杂度

$P(Y^s|x^m,\lambda)$的计算复杂度为$O(1)$

集合大小$S = (N-1)L^2 + NL$

$O(g[]+=)$未变，仍为$O(T)$

所以总体时间复杂度是$O(TNL^2) + O(NL^2) = O(TNL^2)$，比暴力求解$O(TNL^N)$复杂度明显优越。

后面做预测时用的**Viterbi**算法和**Forward-Backward**算法很类似，只是将递推公式中的$\sum$换成Max，时间复杂度是$O(NL^2)$，如果用暴力方法预测，则时间复杂度为$O(TNL^N)$。

为了便于后续对着代码解释，这里将导数公式整理下，

$$
\begin{cases}
\dfrac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j} = \displaystyle\sum_m \left(\sum_i s_j(y_i^m, x^m, i) \right) + \sum_m \left(\sum_i \left(\sum_l s_j(l, x^m, i) P(S_{y_i=l} | x^m, \lambda) \right) \right) \\
\qquad = \displaystyle\sum_m \left(\sum_i s_j(y_i^m, x^m, i) \right) + \sum_m \left(\sum_i \left(\sum_l s_j(l, x^m, i) \frac{1}{Z(x^m)} \alpha_n(a) \beta_n(a) \right) \right) \quad \text{状态特征} \\
\dfrac{\partial \mathcal{L}(\lambda)}{\partial \lambda_j} = \displaystyle\sum_m \left(\sum_i \left(t_j(y_{i-1}^m, y_i^m, x^m, i) \right) \right) + \sum_m \left(\sum_i \left(\sum_{l,l'} \left(t_j(l, l', x^m, i) P(S_{y_{i-1}=l, y_i=l'} | x^m, \lambda) \right) \right) \right) \\
\qquad = \displaystyle\sum_m \left(\sum_i t_j(y_{i-1}^m, y_i^m, x^m, i) \right) + \sum_m \left(\sum_i \left(\sum_{l,l'} t_j(l, l', x^m, i) \frac{1}{Z(x^m)} \alpha_n(a) M_n(l, l') \beta_n(a) \right) \right) \quad \text{转移特征}
\end{cases} \tag{8}
$$

$S_{y_i=l}$，$S_{y_{i-1}=l, y_i=l'}$ 的定义见上文。

Unconstrained Optimization

无约束优化问题数学描述如下：

$$min_x \ f(x)$$

其中$x \in R^n$即n维实数，$f: R^n \rightarrow R$是一个光滑函数, 由于$max_x f(x) = min_x -f(x)$, 所以我们只讨论最小化问题。

一些无约束优化问题有Closed-Form Solution，这样就可以直接套公式求出解，比如$min_x \ (x-a)^2$的解为$x=a$。对于没有闭包形式解的函数，或者闭包形式的解计算和存储代价比较大的时候，就要寻求其他方法。

一般可以通过给定一个初始x_0, 然后不断迭代，搜索一系列的$x_0, x_1, \ldots, x_k$，直到我们认为x_k已经近似接近最优解，或者$f(x_k)$已经足够接近最优值，或者达到我们心里预期值，或者我们的算法在x_k时由于迭代条件无法满足，无法继续迭代了（当然这时也可以换个初始点开始新一轮迭代）。

如果$f(x)$是个强凸函数，由凸函数性质知道$f(x)$存在唯一最小值，并且我们可以通过一定准则判断是否达到或者接近最小值或者知道接下来进一步优化性价比比较低了，比如可以判断导数是否足够接近0。对于非凸函数，如果我们能推出$f(x)$在定义域内有下界，记为$Bound$，也可以根据$f(x_k) - Bound$大小来判断x_k是否已经接近全局最优值，或者根据梯度判读是否接近局部最优值。如果非凸函数没有其他已知信息，就很难找到一个合理的判定准则，但可以根据已知问题领域的知识，估计出一个可以接受的值$Accept$, 根据$|f(x_k) - Accept|$来判定是否已经达到预期优化目标。

Line Search

Line Search[8]顾名思义就是沿着一个线寻找下一个x_{k+1}， 即

$$x_{k+1} = x_k + \alpha \, p_k$$

其中x_k是当前已搜索到的点，p_k是搜索方向，即一个向量，$\alpha \in R^+$，即一维正整数，是待求

解的值，通常称为步长，$x_k + \alpha p_k$构成一条直线，通常限定p_k为函数下降方向，即$f'(x_k)p_k < 0$，这样能保证存在$\alpha > 0$使$f(x_k + \alpha p_k) < f(x_k)$。

我们期望找到一个α，使$f(x_k + \alpha p_k)$相比$f(x_k)$有足够充分的下降。通过将搜索方向限定在一条直线上，我们得到

$$h(\alpha) = f(x_k + \alpha p_k)$$

$h(\alpha)$是关于α的一维函数，$h(\alpha)$有时会比$f(x)$简化很多，存在闭包解。但很多时候$h(\alpha)$也不存在闭包解。这时可以通过一定策略用相对小的代价搜索一个相对比较优的解, 怎么来判定所找到的α已经比较优呢，有很多种判定标准如**Strong Wolfe Conditions**,**Goldstein Conditions**这里只介绍Strong Wolfe Conditions， 因为CRF++代码里面用的是Strong Wolfe Conditions，并且Strong Wolfe Conditions在Quasi-Newton Method中更常用[2]。

Strong Wolfe Conditions

Strong Wolfe Conditions包含两个条件：**sufficient decrease condition**和**curvature condition**

sufficient decrease condition要求函数有足够充分的下降即

$$f(x_k + \alpha p_k) <= f(x_k) + c_1 \alpha f'(x_k)^T p_k$$

也可以记为

$$h(\alpha) <= h(0) + c_1 \alpha h'(0)$$

sufficient decrease condition即要求函数$h(\alpha)$位于直线$f(x_k) + c_1 \alpha f'(x_k)^T p_k$下方，如果$c_1 <= 0$，那这个约束条件就起不到限制$h(\alpha)$下降的目的。如果$c_1 >= 1$那可能不存在满足这个约束条件的解（强凸函数便是一个例子），所以一般要求$0 < c_1 < 1$。

curvature condition要求

$$\left|f'(x_k + \alpha p_k)^T p_k\right| <= c_2 \left|f'(x_k)^T p_k\right|$$

也可以记为

$$\left|h'(\alpha)\right| <= c_2 \left|h'(0)\right|$$

curvature condition要求α处斜率$h'(\alpha)$位于区间$\left[-c_2 \left|h'(0)\right|, c_2 \left|h'(0)\right|\right]$。即期望找到一个导数相比当前导数绝对值足够小的点，因为函数的局部最小值处导数为0，如果足够小，通常也能说明足够接近局部最小值。如果α处导数绝对值比较大，那么**case 1**：导数为负，说明还有明显的进一步下降空间，**case 2**：导数为正说明步子夸大了，把一个局部极小值跨过去了。所以要求$c_2 < 1$即比当前导数绝对值小。如果$c_2 = 0$那约束条件就限制α只能取一些驻点，对于没有驻点的函数，那就找不到满足这个约束条件的解了，即使对于有驻点的函数，也很难找到驻点（如果能方便找到就没必要费力Line Search了），所以要求$c_2 > 0$。综合上述要求$0 < c_2 < 1$。

可以参考Numerical optimization书本上的插图看看。

下面证明

定理1：存在$\alpha > 0$满足Strong Wolfe Conditions条件。记

$$L(\alpha) = f(x_k) + c_1 \alpha\, h^{'}(0)$$
$$F(\alpha) = h(\alpha) - L(\alpha)$$

$L(\alpha)$是一条直线，并且由$0 < c_1 < 1$，容易证明$L(\alpha)$与$h(\alpha)$在$\alpha > 0$的方向有交点（如果$c_1 <= 0$或者$c_1 >= 1$就可能不存在交点），即存在$\alpha > 0$使$F(\alpha) = 0$, 由于$F(0) = 0$所以$F(\alpha) = F(0)$, 由罗尔定理知， 存在$\alpha_x \in (0, \alpha)$使$F^{'}(\alpha) = 0$,即

$$h^{'}(\alpha_x) - c_1 f^{'}(x_k)^T p_k = 0$$

如果限制$c_1 < c_2$那么有

$$\left|h^{'}(\alpha_x)\right| = \left|c_1 h^{'}(0)\right| < \left|c_2 h^{'}(0)\right|$$

即curvature condition也得到满足，综上得证**当$0 < c_1 < c_2 < 1$时，存在$\alpha > 0$满足Strong Wolfe Conditions条件。**

c_1一般取比较小的值，以使满足sufficient decrease condition条件的区间包含局部最优值，或者近似局部最优值。一般取$c_1 = 10^{-4}$

c_2对于Newton Method或Quasi-Newton Method一般取0.9

现在已经给出一个判定标准Strong Wolfe Conditions来判读当前步长α_k是否满足我们目标，并且已证明存在满足Strong Wolfe Conditions的步长，接下来就该Line Search迭代算法出场了。

Line Search Algorithm

直接贴图。图片来自Numerical optimization[2]，里面的ϕ函数即上面列的h函数

Algorithm 3.5 (Line Search Algorithm).

Set $\alpha_0 \leftarrow 0$, choose $\alpha_{\max} > 0$ and $\alpha_1 \in (0, \alpha_{\max})$;
$i \leftarrow 1$;
repeat
  Evaluate $\phi(\alpha_i)$;
  if $\phi(\alpha_i) > \phi(0) + c_1\alpha_i\phi'(0)$ or $[\phi(\alpha_i) \geq \phi(\alpha_{i-1})$ and $i > 1]$
    $\alpha_* \leftarrow$**zoom**(α_{i-1}, α_i) and **stop**;
  Evaluate $\phi'(\alpha_i)$;
  if $|\phi'(\alpha_i)| \leq -c_2\phi'(0)$
    set $\alpha_* \leftarrow \alpha_i$ and **stop**;
  if $\phi'(\alpha_i) \geq 0$
    set $\alpha_* \leftarrow$**zoom**(α_i, α_{i-1}) and **stop**;
  Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{\max})$;
  $i \leftarrow i + 1$;
end (repeat)

图1[2]Line-Search Bracket stage

Algorithm 3.6 (zoom).

repeat

Interpolate (using quadratic, cubic, or bisection) to find a trial step length α_j between α_{lo} and α_{hi};

Evaluate $\phi(\alpha_j)$;

if $\phi(\alpha_j) > \phi(0) + c_1\alpha_j\phi'(0)$ or $\phi(\alpha_j) \geq \phi(\alpha_{\text{lo}})$

$\alpha_{\text{hi}} \leftarrow \alpha_j$;

else

Evaluate $\phi'(\alpha_j)$;

if $|\phi'(\alpha_j)| \leq -c_2\phi'(0)$

Set $\alpha_* \leftarrow \alpha_j$ and **stop**;

if $\phi'(\alpha_j)(\alpha_{\text{hi}} - \alpha_{\text{lo}}) \geq 0$

$\alpha_{\text{hi}} \leftarrow \alpha_{\text{lo}}$;

$\alpha_{\text{lo}} \leftarrow \alpha_j$;

end (repeat)

图2[2]Line-Search Zoom stage

从上面可以看出Line-Search算法分为两步

Bracket Stage

如图一所示，Bracket Stage或者找到一个满足Strong Wolfe Conditions的α_* ,那整个算法就迭代结束了，或者找到一个区间$(\alpha_{lo}, \alpha_{hi})$使这个区间存在满足Strong Wolfe Conditions的步长，图中写的算法如果上面两种case都找不到就继续循环，实际写代码时，一般会设置个循环次数限制，避免这里耗费太多时间，或者进入死循环，毕竟我们上面只是证明存在满足条件的步长，不代表我们可以在有限步骤内能找到这个满足条件的步长。

Zoom Stage

Zoom Stage会不断缩小搜索区间$(\alpha_{lo}, \alpha_{hi})$，直到找到满足Strong Wolfe Conditions的步长。

用**定理1**一样的证明方法，可以证明如果α_{max}选的足够大，则在调用$Zoom$之前，初始化搜索区间$(0, \alpha_{max})$和后续搜索区间(α_i, α_{max})都存在满足Strong Wolfe Conditions的步长。

下面证明

定理2: Zoom Stage初始搜索区间$(\alpha_{lo}, \alpha_{hi})$和后续迭代更新后的区间都存在满足Strong Wolfe Conditions的步长

显而易见两个步骤的区间都是随着各自循环逐渐减小的，如果能够证明**定理2**，则起码说明我们没有盲目在根本不可能存在满足Strong Wolfe Conditions的区间搜索，并且我们搜索区间逐渐减小，离目标也越来越近了。并且假设统一记上面区间为(a, b)，注意这里a可能大于b，也可能小于b,则可以证明$min(h(a), h(b))$在两个步骤循环里面是单调递减的，即每迭代一次找到一个更小值，这样即使最后没有找到满足Strong Wolfe Conditions的步长，我们也可以返回搜索到的最小值对应步长。

容易证明**Zoom函数第一次传入的搜索区间和后续更新后的搜索区间都同时满足下面两个条件**(这里不证了)：

（1）$h'(a)(b-a) < 0$且a满足sufficient decrease condition，即$F(a) <= 0$

（2）如果b满足sufficient decrease condition则$h(a) < h(b)$

当搜索区间同时满足上面两个条件时，那么存在$\alpha \in (a, b)$满足Strong Wolfe Conditions条件。分两种情况来证明

case 1:

假设b满足sufficient decrease condition， 此时有$h(a) < h(b)$

由$h(a) < h(b)$知存在$c \in (a, b)$使得$h'(c)(b-a) > 0$，联合$h'(a)(b-a) < 0$得到存在$d \in (a, c)$使$h'(d) = 0$， 进而知道沿着点a像d的方向走，能找到$h'(\alpha) = 0$的点，记第一次遇见的$h'(\alpha) = 0$的点为α，则α满足curvature condition。又由$h'(a)(b-a) < 0$知，从a走向α的过程，$h(x)$函数值是单调递减的，所以$h(\alpha) < h(a) < h(b)$， 在联合a, b都满足sufficient decrease condition能推出α也满足sufficient decrease condition。综合以上得出α满足sufficient decrease condition。

case 2:

假设b不满足sufficient decrease condition，那么由$F(b) > 0$和$F(a) <= 0$知道存在$c \in (a, b)$使得$F'(c)(b-a) > 0$，又由$h'(a)(b-a) < 0$和a不满足curvature condition能推出$F'(a)(b-a) < 0$，由导数连续性知道从a朝着c的方向走，能找到第一个$F'(\alpha) = 0$的点，并且这个走的方向，$F(x)$值是单调减少的，所以$F(\alpha) < F(a)$，即α满足sufficient decrease condition条件，并且由$F'(\alpha) = 0$得出α满足curvature condition，从而得证。

Convergence Of Line Search

前面一小节已经提到可以证明，Line Search算法搜索到的函数值是单调递减的，但函数值单调递减不代表x_k能够收敛到驻点（如果是凸函数就是全局最小点），单调递减也可能收敛到其他点，或者如果函数无下界，则x_k也可能发散。

如果目标函数$f(x)$满足Lipschitz continuity，记$cos^2(\theta_k) = \frac{-f^{'}(x_k)p_k}{|f(x_k)|\ |p_k|}$即负搜索方向与导数夹角，则容易证明(证明可以参考Numerical optimization[2])

$$\sum_{k>=0} cos^2(\theta_k)||f^{'}(x_k)||^2$$

上式暗含着$cos(\theta_k)||f^{'}(x_k)|| \to 0$，如果每次能保证每次搜索方向与导数夹角余弦大于某一个正整数（Steepest Descent就是每次夹角余弦为1），则$||f^{'}(x_k)||$收敛到0，即证明了在目标函数满足Lipschitz continuity且搜索方向满足一定条件下，Line Search算法会收敛到一个驻点，如果是凸函数，就收敛到全局最小点。

Gradient Gased Method

基于梯度的优化算法，搜索方向形式为

$$p_k = -Bf^{'}(x)$$

其中B是正定矩阵，这样就保证了搜索方向是函数值下降方向

$$f^{'}(x)p_k = -f^{'}(x)Bf^{'}(x) < 0$$

Steepest Descent方法取$B = I$
Newton法取B为Hessian矩阵的逆
Quasi-Newton方法根据最近n次搜索的函数信息来近似一个Hessian矩阵的逆作为B
下面将详细分析各个方法，以及在Line Search方法下的收敛速度
记x*是搜索方向前方驻点

Steepest Descent

令搜索方向$p_k = -If^{'}(x_k) = -f^{'}(x_k)$就得到Steepest Descent算法，正如其名，$-f^{'}(x_k)$方向是函数局部极小领域内函数值下降最快的方向，假设每次Line Search搜索到的α都是局部最优值即Exact Line Search，则$h^{'}(\alpha) = 0$，即

$$h^{'}(\alpha) = f^{'}(x_k + \alpha\, p_k)^T p_k = -f^{'}(x_{k+1})p_k = p_{k+1}p_k = 0$$

，可以看出相邻搜索方向相互垂直。

如果目标函数的二阶导数Hessian矩阵满足$lI \leq\ G(x) \leq LI$，l, L分别是Hessian矩阵G的最小最大特征值，那么可以证明在Exact Line Search下

$$\|x_{k+1} - x^*\|_Q^2 <= \left(\frac{L-l}{L+l}\right)^2 \|x_k - x^*\|_Q^2$$

也即可以证明Line Search线性收敛，这个具体怎么证明我还没精力去看，只看了pluskid[9]在步长固定的情况下对线性收敛的证明。

Newton Method

记G为目标函数Hessian矩阵，令$p_k = -G^{-1}f^{'}(x_k)$，则得到Newton Method。显然如果G是正定矩阵，则$f^{'}(x_k)p_k < 0$，即p_k是函数下降方向，记x^s为函数二阶泰勒近似的全局最小值，则$p_k = x^s - x_k$，即搜索方向指向x^s，所以对于二次函数一次搜索就找到全局最小值。

假设初始点x_0离x^*足够近，并且G满足Lipschitz continuity，每次步长固定为1，可以证明搜索到的点序列x_k收敛到x^*， 并且收敛速度是二次的，证明详见Numerical optimization[2]。

Quasi-Newton Methods

Newton Method要计算和存储Hessian矩阵，并求Hessian矩阵的逆，所以计算和存储代价比较大，Quasi-Newton Methods通过近似Hessian矩阵的逆来减少计算量

DFP

DFP[3]算法通过最近k次迭代的导数及步长信息来近似计算一个Hessian矩阵的逆$H = G^{-1}$，通过迭代增量更新减小计算代价。迭代更新公式如下：

$$H_{k+1} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{\delta_k \delta_k^T}{y_k^T \delta_k} \tag{9}$$

其中$\delta_k = x_{k+1} - x_k$，$y_k = g_{k+1} - g_k$，$g_k = f^{'}(x_k)$，上面公式可以参考文献[3]看看怎么推出的。也可以参考Numerical optimization[2]的推导方法，即先通过带等式约束的最小化求出G的近似B的迭代更新公式，然后通过Sherman–Morrison formula变化得到H的迭代公式。可以采用两次应用rank-one形式变化公式得到H（可以参考逻辑回归模型和算法文中的变换方法）[10]，也可以采取一次rank-two形式变换公式（可以参考Quasi-Newton methods for minimization里面的变化方法）。

推导都要求H_{k+1}满足$H_{k+1}y_k = \delta_k$且$H_{k+1} = H_{k+1}^T$，因为$G\delta_k = y_k$且，$G^T = G$，H_{k+1}想要近似G^{-1}自然要满足这些性质。

文献[3]证明了在二次正定函数下即$f(x) = f(x_0) + a^T x + x^T Gx$，使用公式（9）生成的$H$，**在Exact Line Search下，经过N次迭代后收敛到G^{-1}**，N是数据的纬度，并且搜索到全局最小值。证明过程如下：

如果H_k是对称正定矩阵，容易证明出H_{k+1}也是对称正定矩阵，文献[3]给出了递推证明，但证明有点问题，正确证明应该是$< p|p >< q|q > - < p|q >^2$ 和$< x|\delta_k >^2$不能同时为0，因为如果$< p|p >< q|q > - < p|q >^2 = 0$，则$x = \lambda y_k$，进而
$< x|\delta_k >=< \lambda y_k|\delta_k >=< \delta_k|G|\delta_k > \neq 0$，所以得证。

上面是在$f(x)$是正定函数和Exact Line Search下证明出H_{k+1}保持正定性，其实**对于$f(x)$是任意普通凸函数的情况，也可以证明在Line Search下（不要求Exact），H_{k+1}保持正定性**，从上面推导可以看出只要$y_k^T \delta_k > 0$即可推出H_{k+1}保持正定性，由sufficient decrease condition知$f^{'}(x_{k+1})^T p_k > c_2 f^{'}(x_k)^T p_k$，由$\delta_k = \alpha_k p_k$知$f^{'}(x_{k+1})^T \delta_k > c_2 f^{'}(x_k)^T \delta_k > f^{'}(x_k)^T \delta_k$，进而推出$(f^{'}(x_{k+1}) - f^{'}(x_k))^T \delta_k > 0$即$y_k^T \delta_k > 0$。

H_k正定保证了搜索方向下降。

如果能够证明$H^N G\delta_i = \delta_i$ ，$0 \leq i < N$，则易知$H^N = G^{-1}$，**即H经过N次迭代收敛到G^{-1}**

接下来证明下面一组公式

$$\delta_i^T G \delta_j = 0 \ , 0 \le i < j < k \tag{10}$$

$$H_k G \delta_i = \delta_i \ , 0 \le i < k \tag{11}$$

当$k=1$，显然成立，下面递推证明，假设已知k时，上面一组公式成立，则

$$g_k = a + Gx_k = a + G(x_{i+1} + \delta_{i+1} + \ldots + \delta_{k-1}) = g_{i+1} + G(\delta_{i+1} + \ldots + \delta_{k-1})$$

$$\delta_i^T g_k = \delta_i^T g_{i+1} + \delta_i^T G(\delta_{i+1} + \ldots + \delta_{k-1}) = \delta_i g_{i+1} = 0 \ , 0 \le i < k$$

联合上面公式和公式(11)可以得到

$$0 = \delta_i^T g_k = -\alpha_k \delta_i^T G H_k g_k = -\alpha_k \delta_i^T G \delta_k \ , 0 \le i < k$$

上式和k公式(10)合并起来即知$k+1$时，公式(10)仍然成立。

$$H_{k+1} G \delta_i = H_k G \delta_i - \frac{H_k y_k y_k^T H_k G \delta_i}{y_k^T H_k y_k} + \frac{\delta_k \delta_k^T G \delta_i}{y_k^T \delta_k} = H_k G \delta_i - \frac{H_k y_k y_k^T H_k G \delta_i}{y_k^T H_k y_k} + \frac{\delta_k \delta_k^T G \delta_i}{y_k^T \delta_k} \ , 0 \le i < k+1$$

由$k+1$时的公式(10)和k时公式(11)可以推出

$$H_{k+1} G \delta_i = H_k G \delta_i - \frac{H_k y_k y_k^T \delta_i}{y_k^T H_k y_k} + \frac{\delta_k \delta_k^T y_i}{y_k^T \delta_k} = H_k G \delta_i - \frac{H_k y_k (G\delta_k)^T \delta_i}{y_k^T H_k y_k} + \frac{\delta_k \delta_k^T G \delta_i}{y_k^T \delta_k} = H_k G \delta_i - 0 - 0 = \delta_i \ , 0 \le i < k+1$$

即得证$k+1$时公式(11)成立。从而证明了在二次正定函数下，经过N次迭代后收敛到Hessian矩阵G的逆。

BFGS

BFGS与DFP方法类似，迭代公式推导方法也类似。

$$min_H \ \|H - H_k\| \ , subject \ to \ H = H^T \ , Hy_k = s_k$$

其中s_k即为DFP中的δ_k，通过最小化上面式子即求出H_k的迭代更新公式

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$$

$$\rho_k = \frac{1}{y_k^T s_k}$$

据说特定条件下DFP和BFGS都能保证全局收敛性，并且都具有超线性收敛速度，但in practice BFGS比DFP更鲁棒高效，所以使用更普遍。

L-BFGS

DFP和BFGS要计算和存储H，存储和迭代计算H的代价都是$O(N^2)$。在大型广告和推荐系统中，N可能达到上亿级别，这样存储和计算代价太高。根据H_k的递推公式，将H_k做m次展开，得

$$
\begin{cases}
H_k = V_{k-1}^T V_{k-2}^T \dots V_{k-m}^T H_{k-m} V_{k-m} \dots V_{k-2} V_{k-1} \\
\quad + V_{k-1}^T V_{k-2}^T \dots V_{k-m+1}^T s_{k-m} \rho_{k-m} s_{k-m}^T V_{k-m+1} \dots V_{k-2} V_{k-1} \\
\quad + V_{k-1}^T V_{k-2}^T \dots V_{k-m+2}^T s_{k-m+1} \rho_{k-m+1} s_{k-m+1}^T V_{k-m+2} \dots V_{k-2} V_{k-1} \\
\quad + \dots \\
\quad + V_{k-1}^T s_{k-2} \rho_{k-2} s_{k-2}^T V_{k-1} \\
\quad + s_{k-1} \rho_{k-1} s_{k-1}^T
\end{cases} \tag{12}
$$

其中$V_k = I - \rho_k y_k s_k^T$

如果取H_{k-m}为对角阵，则H_{k-m}也可以当做一个$O(N)$的向量存储，或者可以将H_{k-m}分解为两个向量的乘积形式。相当于将H_k全部用向量表示出来，V_k也是用向量表示的。用最近m次搜索的函数信息来近似H_k。这样存储代价变为$O(Nm)$。

我们最终目的是通过$p_k = -H_k g_k$求出迭代搜索方向。如果将g_k从右向左，右乘以H_k展开式，由于**都是向量点击运算**，（因为上文已知H_k展开式里面每个元素都是向量，g_k右乘H_k遇见的每个向量或者是航向量，这是就是向量点击；或者遇见列向量，这时已经乘出来的部分就是个常数），所以**计算代价是$O(Nm^2)$**，观察H_k展开式可以看出，里面**存在可以复用的子运算**结构。比如H_k展开式每行都是对称结构，相邻两行间大部分运算也是一样的。

记$q_{k-m+1} = V_{k-m+2} \dots V_{k-2} V_{k-1} g_k$表示$H_k$展开式某一行（其实是第三行）的右边，则易知

$\alpha_{k-m+1} = \rho_{k-m+1} s_{k-m+1}^T q_{k-m+1}$

$q_{k-m} = q_{k-m+1} - \alpha_{k-m+1} y_{k-m+1} = V_{k-m+1} q_{k-m+1}$（第二行右边）

即α, q存在递推关系，从下往上，可以将$H_k g_k$展开式每一行右边递推出来，存到临时变量α, q。

观察H_k展开式左边，可以发现前r行的左边前$m - r + 1$项是完全一样的，即可以将前r行的左边部分提取出作为公共乘子。另一方面我们需要将所有行累加起来，所以可以采取从上到下迭代累加方式。记r_r表示前r行扣除左边最长公共乘子的累加和，

$r_1 = H_{k-m} q_{k-m-1} = H_{k-m} V_{k-m} \dots V_{k-2} V_{k-1}$表示前1行扣除公共乘子的累加和，则

$$
\begin{aligned}
r_2 &= V_{k-m}^T r_1 + s_{k-m} \alpha_{k-m} \\
r_3 &= V_{k-m+1}^T r_2 + s_{k-m+1} \alpha_{k-m+1} \\
r_{r+1} &= V_{k-m+r-1}^T r_r + s_{k-m+r-1}^T \alpha_{k-m+r-1}
\end{aligned}
$$

记$\beta_{r+1} = \rho_{k-m+r-1} y_{k-m+r-1}^T r_r$，则

$$
r_{r+1} = r_r + s_{k-m+r-1} (\alpha_{k-m+r-1} - \beta_{r+1})
$$

易知$r_{m+1} = H_k g_k$，且上面总的**计算时间复杂度是$O(Nm)$**。将上述计算过程描述出来即如下

图所示（图片来自Numerical optimization[2]，∇f_k即g_k）

Algorithm 7.4 (L-BFGS two-loop recursion).

$q \leftarrow \nabla f_k;$
for $i = k-1, k-2, \ldots, k-m$
$\alpha_i \leftarrow \rho_i s_i^T q;$
$q \leftarrow q - \alpha_i y_i;$
end (for)
$r \leftarrow H_k^0 q;$
for $i = k-m, k-m+1, \ldots, k-1$
$\beta \leftarrow \rho_i y_i^T r;$
$r \leftarrow r + s_i(\alpha_i - \beta)$
end (for)
stop with result $H_k \nabla f_k = r.$

IIS

IIS即 Improved Iterative Scaling Algorithm的思想如下，对于求解凹函数$f(x)$的最大值问题，如果可以找到一个新的更简单的凹函数$g(x)$使得

$$\begin{cases} g(x) & \geq f(x) \\ g(x_k) & = f(x_k) \\ g'(x_k) & = f'(x_k) \end{cases} \tag{13}$$

那么对$f(x)$的迭代求最小值可以转化为对$g(x)$迭代求最小值（$g(x)$可能需要在每次迭代后重新构造）。因为
case 1：如果$g(x)$能找到一个$g(x_{k+1}) > g(x_k)$，那么由公式（13）知道

$$f(x_{k+1}) \geq g(x_{k+1}) > g_k \Rightarrow f(x_{k+1}) > f(x_k)$$

即$g(x)$的下一个搜索到的更大值位置x_{k+1}也使$f(x)$的值增大。
case 2：如果$g(x)$无法找到一个比$g(x_k)$更大的值，那么由$g(x)$是凹函数，可知$g'(x_k) = 0$，又由公式（13）知$f'(x_k) = 0$所以$f(x_k)$也是$f(x)$的最大值，求解可以结束了。

问题主要是怎么构造满足公式（13）的凹函数$g(x)$，使$g(x)$或者$g'(x)$足够简单方便求解。比如x是多维向量，如果可以找到一个新的$g(x)$，使$g(x)$可以分解成各个独立无关的部分，每个部分只含其中一个维度的变量。或者使$\frac{\partial g(x)}{\partial x_j}$只含单一维度变量。那找$g(x)$的更小值就简单很多。

对于类似CRF这种形式的目标函数$\mathcal{L}(\lambda)$（参见公式（3））， 文献[5]阐述了一种构造求解方法。

记λ^k是当前搜索到的最优点，$f(\delta) = \mathcal{L}(\lambda^k + \delta) - \mathcal{L}(\lambda^k)$，则最大化$f(\delta)$与最大化$\mathcal{L}(\lambda)$是

等价的。$\lambda^* = \lambda^k + \delta^*$，其中$\lambda^k, \delta^*$分别是$\mathcal{L}(\lambda), f(\delta)$的最大值。

$$f(\delta) = \sum_m \left(\sum_j \delta_j F_j \left(x^m, y^m\right) - log \frac{Z_{\lambda^k+\delta}\left(x^m\right)}{Z_{\lambda^k}\left(x^m\right)} \right)$$

下面证明一个不等式

$$-log\left(x\right) \geq 1 - x,\ x \in (0,1)$$

记$LogF(x) = -log(x) - 1 + x$，则$LogF(1) = 0$且$LogF^{'}(x) = 1 - \frac{1}{x} <= 0,\ x \in (0,1)$，由$LogF(x)$在$(0,1)$区间单调下降，且$LogF(1) = 0$，推出$LogF(x) > 0,\ x \in (0,1)$，上面不等式得证。

利用上面不等式得

$$\begin{cases} f(\delta) & \geq \sum_m \left(\sum_j \delta_j F_j \left(x^m, y^m\right) + 1 - \frac{Z_{\lambda^k+\delta}\left(x^m\right)}{Z_{\lambda^k}\left(x^m\right)} \right) \\ & = \sum_m \left(\sum_j \delta_j F_j \left(x^m, y^m\right) + 1 - \sum_y p_{\lambda^k(y|x^m)} exp\left(\sum_j \delta_j F_j(x^m, y) \right) \right) \end{cases} \tag{14}$$

记$F^{\#}(x,y) = \sum_j F_j(x,y)$，知$\sum_j \frac{F_j(x^m,y)}{F^{\#}(x^m,y)} = 1$，应用Jensen's inequality，得

$$\begin{cases} f(\delta) & \geq \sum_m \left(\sum_j \delta_j F_j \left(x^m, y^m\right) + 1 - \sum_y p_{\lambda^k(y|x^m)} exp\left(\sum_j \delta_j F^{\#}(x^m,y) \frac{F_j(x^m,y)}{F^{\#}(x^m,y)} \right) \right) \\ & \geq \sum_m \left(\sum_j \delta_j F_j \left(x^m, y^m\right) + 1 - \sum_y p_{\lambda^k(y|x^m)} \sum_j \left(\frac{F_j(x^m,y)}{F^{\#}(x^m,y)} exp\left(\delta_j F^{\#}(x^m,y)\right) \right) \right) \end{cases} \tag{15}$$

记上式右边为$g(\delta)$，则$g(\delta)$是凹函数且满足公式（13）的3个条件。可以验证$g(0) = f(0)$且$g^{'}(0) = f^{'}(0)$,有上面不等式知$f(\delta) \geq\ g(\delta)$。

现在只要找出一个$g(\delta) > g(0)$即可，

$$\frac{\partial\, g(\delta)}{\partial \delta_j} = \sum_m \left(\sum_j F_j \left(x^m, y^m\right) + 1 - \sum_y p_{\lambda^k(y|x^m)} F_j(x^m,y) exp\left(\delta_j F^{\#}(x^m,y)\right) \right)$$

导数为0的点即为$g(\delta)$最大值的点，上面梯度只含单一维度变量，令$\frac{\partial\, g(\delta)}{\partial \delta_j} = 0$可以直接用牛顿法求出各个$\delta_j$，并且求解速度很快，二次收敛速度。

代码分析

CRF代码分析

关键函数执行流程如下，附注释

```
Encoder::learn(const char *templfile,//模板文件，用于生成特征，包括Unigram（状态特征）和Bigram（转移特征）
                    const char *trainfile,//训练文件，里面有(x,y)标注样本
                    size_t maxitr,//最大迭代次数，这里相当于求函数梯度次数，当迭代次数超过maxitr时就结束继续优化
                    size_t freq,//特征最小出现频次，小于此值则过滤掉
                    double eta, //连续3次相邻两次迭代似然函数值之差小于eta的话，就结束继续优化
                    double C,//L2正则项权重缩放因子
                    unsigned short thread_num//求梯度并发线程数
                    )
{
    EncoderFeatureIndex feature_index;
    feature_index.open(templfile, trainfile))//读入模板文件， 根据训练文件提取标注集合，即看看y可能取哪些标注
  std::ifstream ifs(trainfile);
   while (ifs) {
      TaggerImpl *_x = new TaggerImpl();
      _x->open(&feature_index, &allocator);
      if (!_x->read(&ifs) || !_x->shrink()) {//读入训练样本， 每个TaggerImpl对应一个训练样本；先TaggerImpl::read读入一个样本及对应的标注序列，
          //并调用TaggerImpl::shrink()生成特征，分配特征id（特征字符串一样，则id一样，同时记录每个特征出现的频次）；每个样本每个位置的特征id集合打包成一个vector保存在 FeatureCache中；这里一个id是只根据x，未用y（标注序列）生成的，所以每个id如果是Unigram生成的，实际上对应L个连续的id，L是标注集合大小，即标注取值个数，每个id如果是Bigram生成的则对应L*L个连续的id，即当相邻的y_(i-1),y_i可能取值个数。
      }
      _x->set_thread_id(line % thread_num);//这个样本分配给第(line % thread_num)个线程处理
    }
  feature_index.shrink(freq, &allocator); //所有训练数据读完后，根据频次统计，将低频特征去掉， 并重新分配特征id，以使特征空间紧凑；即一些特征id丢掉后，特征id会出现不连续情况，所以重新分配id，以使max(id)以内的所有id都有用。
  //因为后面优化过程中分配的空间跟max(id)有关系,FeatureIndex::size() { return maxid_; }
  //样本数据(x,y)都读进来了，状态和转移特征都生成了，下面可以跑优化，求解每个特征对应的权重了
  std::vector <double> alpha(feature_index.size()); //alpha即待求解的特征权重数组
  runCRF(x, &feature_index, &alpha[0],
                  maxitr, C, eta, shrinking_size, thread_num, false);
}
```

```
bool runCRF(const std::vector<TaggerImpl* > &x,
            EncoderFeatureIndex *feature_index,
            double *alpha,
            size_t maxitr,
            float C,
            double eta,
            unsigned short thread_num) {
  double old_obj = 1e+37;
  int    converge = 0;
  LBFGS lbfgs;
  std::vector<CRFEncoderThread> thread(thread_num);
  //这里线程初始化，代码省略
  for (size_t itr = 0; itr < maxitr; ++itr) {
     //这里启动CRFEncoderThread线程，代码省略,启动后，CRFEncoderThread::run()(详见下一个代码注释)得到执行，
     //这里等待所有CRFEncoderThread线程计算完毕，多个线程正忙碌着跑forward-backward算法计算梯度thread[0].expected，也同时计算似然函数值thread[0].obj，并用当前特征权重跑viterbi算法预测每个样本的标注序列，然后与样本的真实标注序列比对，记录预测错误数到thread[0].err和thread[0].zeroone
     //这里所有线程已经计算完毕，下面将所有线程的结果合并
     for (size_t i = 1; i < thread_num; ++i) {
        thread[0].obj += thread[i].obj;//将函数值累加
        thread[0].err += thread[i].err;//预测错误的的标注数，一个样本的一个位置预测错则计数一次
        thread[0].zeroone += thread[i].zeroone;//预测错误的样本（序列）数，整个样本只要有一个标注预测错则计数一次，多个标注错也只计数一次
     }
     for (size_t i = 1; i < thread_num; ++i) {
        for (size_t k = 0; k < feature_index->size(); ++k) {
            thread[0].expected[k] += thread[i].expected[k];//所有线程梯度值累加起来
        }
     }
     for (size_t k = 0; k < feature_index->size(); ++k) {
         thread[0].obj += (alpha[k] * alpha[k] //(2.0 * C));//函数值加上L2项
         thread[0].expected[k] += alpha[k] //梯度值加上L2项
      }
      //判断相邻两次迭代的目标函数值之差
      double diff = (itr == 0 ? 1.0 : std::abs(old_obj -
         thread[0].obj)/old_obj);
      old_obj = thread[0].obj;
      converge += (diff < eta) ? 1 : 0;
      //如果连续3次目标函数值diff小于阀值，或者达到最大迭代次数，则停止迭代
      if (itr > maxitr || converge == 3) {
          break;
      }
      //lbfgs里面iflag=0（gnorm / xnorm <= eps 时）会导致这里返回值<=0，被认为失败， 感觉应该认为是已经足够接近最优值才对。
      if (lbfgs.optimize(feature_index->size(),
                         &alpha[0],
```

```
                        thread[0].obj,
                        &thread[0].expected[0], orthant, C) <= 0) {
          return false;
        }
    }//itr < maxitr
    return true;
}
```

```
void CRFEncoderThread::run() {
    obj = 0.0; err = zeroone = 0;
    std::fill(expected.begin(), expected.end(), 0.0);//梯度初始化为0
    for (size_t i = start_i; i < size; i += thread_num) {
      obj += x[i]->gradient(&expected[0]);//obj为似然函数值，x即样本，对每个样本调用TaggerImpl::gradient(double *expected)算梯度，并跑viterbi算法预测标注序列
      int error_num = x[i]->eval();//评估预测值与真实标注的误查
      err += error_num;//预测错误的的标注数，一个样本的一个位置预测错则计数一次
      if (error_num) {
        ++zeroone;//预测错误的样本（序列）数，整个样本只要有一个标注预测错则计数一次，多个标注错也只计数一次
      }
    }
  }
```

接下来就是比较重要的**计算梯度**的代码，这里可以对照公式（8）来看，进入这个函数已经确定样本了，即最外面的累加m下标已经确定。

先看公式的前半部分，即经验分布期望。对于状态特征，node_[i][answer_[i]等于y_i^m即第m个样本在第i个位置的真实标注值，node_[i][answer_[i]->fvector存放着这个位置标注确定后，对应的所有状态特征id，因为一个位置可能有多个Unigram模板提取出来的不同特征。转移特征经验分布类似。

再看公式的后半部分，即模型分布期望。下面代码的i对应着公式里面的i，即序列下标。node_[i][j]对应着l(状态特征)，Node::calcExpectation里面的lpath对应着l,l'转移特征。公式（8）只是对其中一个特征做计算，这里要对所有特征计算，因为一旦i确定后，这个序列位置i的所有特征的$\frac{1}{Z(x^m)}\alpha_n(a)\beta_n(a)$值都是一样的，所以把这个模型概率加到所有特征id即可（因为特征函数返回0或1）。转移特征模型分布也类似。

TaggerImpl::gradient()函数返回似然函数值Z_ - s ，可以对照公式（3）的来看怎么计算的。

上面的梯度和函数值计算时是把公式前面加个负号，将最大化问题转化为最小化问题。

buildLattice()可以参考hankcs的CRF++代码分析里面的图

注意计算前向后向概率时为避免浮点溢出，采用将值取对数保存方式，即logsumexp方式，详见hankcs的计算指数函数的和的对数

```
double TaggerImpl::gradient(double *expected) {
  buildLattice();//构建Node 和 Path ： 并计算Node（对于状态特征）和Path权重（对应转移特征）
  forwardbackward();//根据公式（7）递推计算前向概率alpha和后向概率beta
  double s = 0.0;
  for (size_t i = 0;   i < x_.size(); ++i)
    for (size_t j = 0; j < ysize_; ++j)
      node_[i][j]->calcExpectation(expected, Z_, ysize_); //计算梯度中的状态特征模型分布期望，和转移特征模型分布期望。
  for (size_t i = 0;   i < x_.size(); ++i) {
    for (const int *f = node_[i][answer_[i]]->fvector; *f != -1; ++f) {
      --expected[*f + answer_[i]];//梯度中的状态特征经验分布期望，对应状态特征，一个f实际上对应L个特征，即f是根据样本x值生成的，未考虑y，对应不同的y就有不同的特征，f只是这一组特征的起始值，所以这里加上answer_[i]，即实际样本真实标注的特征。
    }
    s += node_[i][answer_[i]]->cost;  // UNIGRAM cost公式（3）似然函数值里面的状态特征权重
    const std::vector<Path *> &lpath = node_[i][answer_[i]]->lpath;
    for (const_Path_iterator it = lpath.begin(); it != lpath.end(); ++it)
 {
      if ((*it)->lnode->y == answer_[(*it)->lnode->x]) {//判断当前path是否是样本真实标注值对。
        for (const int *f = (*it)->fvector; *f != -1; ++f) {
          --expected[*f +(*it)->lnode->y * ysize_ +(*it)->rnode->y];//梯度中的转移特征经验分布期望，这里加(*it)->lnode->y * ysize_ +(*it)->rnode->y的含义与状态特征类似，只不过这里一个f对应L^2个特征。
        }
        s += (*it)->cost;  // BIGRAM COST公式（3）似然函数值里面的转移特征权重
        break;
      }
    }
  }
  viterbi();  // call for eval()  跑viterbi算法，用当前特征权重预测标注序列y，后面会那预测值于真实标注值比较，以评估当前特征权重的模型好坏。
  return Z_ - s ;//返回似然函数值，即公式（3）的值，取负即可
}
```

Node::calcAlpha()和Node::calcBeta()可以参考公式（7）递推公式，
注意Node::calcBeta()与公式（7）有点不同，最后多加了一项beta += cost; 等于比公式（7）中的值多了第一个节点的状态特征权重。所以后续使用时也稍微变化了下

```
void Node::calcAlpha() {
  alpha = 0.0;
  for (const_Path_iterator it = lpath.begin(); it != lpath.end(); ++it) {
    alpha = logsumexp(alpha,
                      (*it)->cost +(*it)->lnode->alpha,
                      (it == lpath.begin()));
  }
  alpha += cost;//前向后向概率保存的是取对数后的值，防止保存原始值时中间计算溢出； log(....)
}
void Node::calcBeta() {
  beta = 0.0;
  for (const_Path_iterator it = rpath.begin(); it != rpath.end(); ++it) {
    beta = logsumexp(beta,
                     (*it)->cost +(*it)->rnode->beta,
                     (it == rpath.begin()));
  }
  beta += cost;//这里比公式（7）多加了第一个节点的状态特征权重
}
void Node::calcExpectation(double *expected, double Z, size_t size) const {
  const double c = std::exp(alpha + beta - cost - Z);//上面beta多加了一个cost，所以这里减掉
  for (const int *f = fvector; *f != -1; ++f) {
    expected[*f + y] += c;    //状态特征期望
  }
  for (const_Path_iterator it = lpath.begin(); it != lpath.end(); ++it) {
    (*it)->calcExpectation(expected, Z, size);//计算转移特征期望
  }
}
```

L-BFGS代码分析

带L1正则化case我还没看，这里就忽略掉了

```
void LBFGS::lbfgs_optimize(int size, //数组x和g的长度
                           int msize,//即lbfgs里面的m，使用最近m次迭代的函数信息
                           double *x,//当前x值，对于crf++就是特征权重数组
                           double f,//当前函数值，对于crf++就是似然函数值
                           const double *g,//当前梯度
                           double *diag,//存放H(-m)即初始对角阵

                           double *w,//存放最近m次迭代的ρ，y，s，以及计算搜索方向时用来存放临时变量α，q，r，等
                           double *v,//对于L2正则化v=g
                           int *iflag) {
  double yy = 0.0;
  double ys = 0.0;
  int bound = 0;
  int cp = 0;
  if (*iflag == 1) goto L172;//*iflag为1意味着上次执行了Line search(mcsrch)，并且未找到满足Wolfe condition的step，但计算出一个新的step，并同时更新了x数组，返回到CRF主程序，让CRF计算step步长时的函数值和梯度值，执行到这里时，说明CRF计算完毕，重新进来直接跳到L172，让mcsrch评估当前步长是否满足Wolfe condition
  if (*iflag == 2) goto L100;//这个分支代码里未用，感觉是由用户自己生成diag（H(-m)）来用
  //执行到这里意味着iflag_ = 0，iflag_只会出现一次，只有第一次调用lbfgs时才为0
  //w_.resize(size * (2 * msize + 1) + 2 * msize);
  //w[0]未用
  //w[ (1, size) ] 存放lbfgs迭代时的临时变量q或r；在计算出line search搜索方向后，w[ (1, size) ]也用来存放梯度f'(x_k)，以便后面确定好步长后，来计算梯度变化（w[iypt + npt + i] = g[i] - w[i];）注意每次传进来的g可能不是f'(x_k)，也有可能是f'(x_k+step*p_k)
  //w[(size + 1, size + msize)] 存放ρ
  //w[(size + msize + 1, size + 2*msize)] 存放α
  //w[(size + 2*msize + 1, size + 2*msize+msize*size)] 存放s;  初始化s存储区域基址；ispt = size + (msize << 1);
  //w[(size + 2*msize + msize*size +1, size + 2*msize+2*msize*size)] 存放y;初始化y存储区基址；iypt = ispt + size * msize;
  //第一次调用lbfgs，做初始化
  if (*iflag == 0) {
    point = 0;
    for (int i = 1; i <= size; ++i) {
      diag[i] = 1.0;//H（-m）初始化为单位矩阵
    }
    ispt = size + (msize << 1);//s下标基址
    iypt = ispt + size * msize;//y下标基址
    for (int i = 1; i <= size; ++i) {
      w[ispt + i] = -v[i] * diag[i];//初始化第一次迭代搜索方向
    }
    stp1 = 1.0 / std::sqrt(ddot_(size, &v[1], &v[1]));
  }
  // MAIN ITERATION LOOP
  while (true) {
  //执行到这里，或者是从第一次调用lbfgs进来的（iflag=0），或者就是从while循环下面mc
```

```
srch之后跳转过来的，mcsrch找到满足Wolfe condition的step后，会继续下一轮循环，跑到这里以重新用lbfgs计算新的迭代方向。所以也意味着运行到这里后，下次传给mcsrch的是新的搜索方向。
    ++iter;
    info = 0;//每次计算出新的搜索方向，info要置为1，这样mcsrch知道传进来的参数是f(x_k),f'(x_k),mcsrch会把这个值保存下来，后续跟其他f(x_k+step*p_k),f'(x_k+step*p_k)比较，判断是否满足Wolfe condition
    if (iter == 1) goto L165;//第一次迭代，没有历史s，y等搜索信息，所以直接用上面iflag=0计算的初始搜索方向
    ys = ddot_(size, &w[iypt + npt + 1], &w[ispt + npt + 1]);
    yy = ddot_(size, &w[iypt + npt + 1], &w[iypt + npt + 1]);
    //初始化diag=H(-m)
    for (int i = 1; i <= size; ++i) {
      diag[i] = ys / yy;
    }
 L100:
    cp = point;
    if (point == 0) cp = msize;
    w[size + cp] = 1.0 / ys; //存放ρ
    //初始化w[(1,size)] 为-g
    for (int i = 1; i <= size; ++i) {
      w[i] = -v[i];
    }
    bound = min(iter - 1, msize); //用最近bound次的迭代信息（ρ，y，s）来求近似-H(-m)*g
    //下面步骤迭代计算-H(-m)*g，可以参考L-BFGS小节LBFGS two loop  recursion那个图
    //迭代计算q和α
    //α= ρ * s_t * q
    //q= q - a*y
    cp = point;//point指向最新的搜索信息s，y的下一个位置，即未来存放s，y的下标，所以下面cp先减一。
    for (int i = 1; i <= bound; ++i) {
      --cp;
      if (cp == -1) cp = msize - 1;
      double sq = ddot_(size, &w[ispt + cp * size + 1], &w[1]);
      int inmc = size + msize + cp + 1;
      iycn = iypt + cp * size;
      w[inmc] = w[size + cp + 1] * sq;
      double d = -w[inmc];
      daxpy_(size, d, &w[iycn + 1], &w[1]);
    }
    //初始化r=-H(-m)*q
    for (int i = 1; i <= size; ++i) {
      w[i] = diag[i] * w[i];
    }
    //迭代更新r
    //beta = ρ_i * y_t * r
    //r = r + (alpha_i - beta) * s
    for (int i = 1; i <= bound; ++i) {
      double yr = ddot_(size, &w[iypt + cp * size + 1], &w[1]);
      double beta = w[size + cp + 1] * yr;
```

```
      int inmc = size + msize + cp + 1;
      beta = w[inmc] - beta;
      iscn = ispt + cp * size;
      daxpy_(size, beta, &w[iscn + 1], &w[1]);
      ++cp;
      if (cp == msize) cp = 0;
    }
    //将搜索方向r=-H*g存放到w[ispt + point * size]
    for (int i = 1; i <= size; ++i) {
      w[ispt + point * size + i] = w[i];
    }
 L165:
    // OBTAIN THE ONE-DIMENSIONAL MINIMIZER OF THE FUNCTION
    // BY USING THE LINE SEARCH ROUTINE MCSRCH
    nfev = 0;//nfev表示mcsrch评估函数值和梯度次数，置为0，每评估一次mcsrch里面会加一，达到最大次数后就认为搜索失败，整个算法求解就失败了，CRF主程序也退出。
    stp = 1.0;
    if (iter == 1) {
      stp = stp1;
    }
    for (int i = 1; i <= size; ++i) {
      w[i] = g[i];//存储梯度f'(x_k)，供后面找到满足Wolfe condition步长后用来计算梯度变化y=f'(x_k+step*p_k)-f'(x_k)
    }
 L172:
 //每次计算出新搜索方向后，第一次调用mcsrch时， info=0， 导数g存于 w[1，size]中，
 //当后续不断迭代mcsrch知直到找出符合Wolfe condition的stp，即当info=1时，这时便可以将导数变化g-w[1,size]存到w[iypt + npt]中，并设置
 //ρ，存于w[size + cp] = 1.0 / ys（详见L100处代码）
 //在找到Wolfe condition步长前，iflag=1，会直接跳转鳌L172．或者达到最大maxfev = 20还未找到合适步长，就认为失败
    mcsrch_->mcsrch(size, &x[1], f, &v[1], &w[ispt + point * size + 1],
                    &stp, &info, &nfev, &diag[1]);
    if (info == -1) {
      *iflag = 1;  //未找到合适步长，就用插值方法计算出新的步长，并更新x数组，返回CRF程序，让CRF计算新的x处的函数值和梯度
      return;
    }
    if (info != 1) {//寻找失败，可能是达到maxfev = 20，也可能是其他情况
      std::cerr << "The line search routine mcsrch failed: error code:"
                << info << std::endl;
      *iflag = -1;
      return;
    }
    //找到了满足Wolfe condition的stp，
    npt = point * size;
    for (int i = 1; i <= size; ++i) {
      w[ispt + npt + i] = stp * w[ispt + npt + i];//计算存储s=stp*p_k=stp*H(-m)*g
      w[iypt + npt + i] = g[i] - w[i];//计算存储y=f'(x_k+step*p_k)-f'(x_k)
    }
    ++point;
```

```
    if (point == msize) point = 0;//保留最近m次搜索的s，y等信息,循环复用数组
    double gnorm = std::sqrt(ddot_(size, &v[1], &v[1]));
    double xnorm = max(1.0, std::sqrt(ddot_(size, &x[1], &x[1])));
    if (gnorm / xnorm <= eps) {//梯度的模足够小就停止继续
      *iflag = 0;  // OK terminated
      return;
    }
    //继续循环，用l-bfgs求下一次的搜索方向。
  }
  return;
  }
```

---

1. Lafferty, J., McCallum, A., Pereira, F. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. Proc. 18th International Conf. on Machine Learning. Morgan Kaufmann: 282–289. 2001. ↩
2. Gilbert, J. C., & Lemarechal, C. (2006). Numerical optimization: theoretical and practical aspects. Springer. ↩
3. R. Fletcher and M. J. D. Powell, A Rapidly Convergent Descent Method for Minimization , Computer Journal 6 (1963), 163–168. ↩
4. J.NOCEDAL, Updating quasi-Newton matrices with limited storage,MathematicsofComputation, 35 (1980), pp. 773–782. ↩
5. Berger A. The improved iterative scaling algorithm: A gentle introduction[J]. Unpublished manuscript, 1997. ↩
6. 统计学习方法，李航 ↩
7. 最大熵模型http://blog.csdn.net/zgwhugr0216/article/details/53240577 ↩
8. Moré J J, Thuente D J. Line search algorithms with guaranteed sufficient decrease[J]. ACM Transactions on Mathematical Software (TOMS), 1994, 20(3): 286-307. ↩
9. pluskid. Gradient Descent, Wolfe’s Condition and Logistic Regression http://freemind.pluskid.org/machine-learning/gradient-descent-wolfe-s-condition-and-logistic-regression/ ↩
10. 逻辑回归模型和算法http://blog.csdn.net/zgwhugr0216/article/details/50145075 ↩