梯度下降和拟牛顿

七月算法 邹博

2015年10月31日

Lagrange对偶函数的鞍点解释

- 为表述方便，假设没有等式约束，只考虑不等式约束，结论可方便的扩展到等式约束。
- 假设x0不可行，即存在某些i，使得$f_i(x)>0$。则选择 $\lambda_i \to \infty$，对于其他乘子，$\lambda_j = 0, j \neq i$
- 假设x0可行，则有$f_i(x)\leqslant 0,(i=1,2,...,m)$,选择

$$\lambda_i = 0, i = 1,2,\cdots,m$$

- 有：

$$\sup_{\lambda \geq 0} L(x,\lambda) = \sup_{\lambda \geq 0}\left(f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) \right) = \begin{cases} f_0(x) & f_i(x) \leq 0, i = 1,2\cdots,m \\ \infty & otherwise \end{cases}$$

鞍点：最优点

- 而原问题是：$\inf_x f_0(x)$
- 从而，原问题的本质为：$\inf_x \sup_{\lambda \geq 0} L(x, \lambda)$
- 而对偶问题，是求对偶函数的最大值，即：

$$\sup_{\lambda \geq 0} \inf_x L(x, \lambda)$$

- 而：

$$\sup_{\lambda \geq 0} \inf_x L(x, \lambda) \leq \inf_x \sup_{\lambda \geq 0} L(x, \lambda)$$

证明：$\max_x \min_y f(x,y) \le \min_y \max_x f(x,y)$

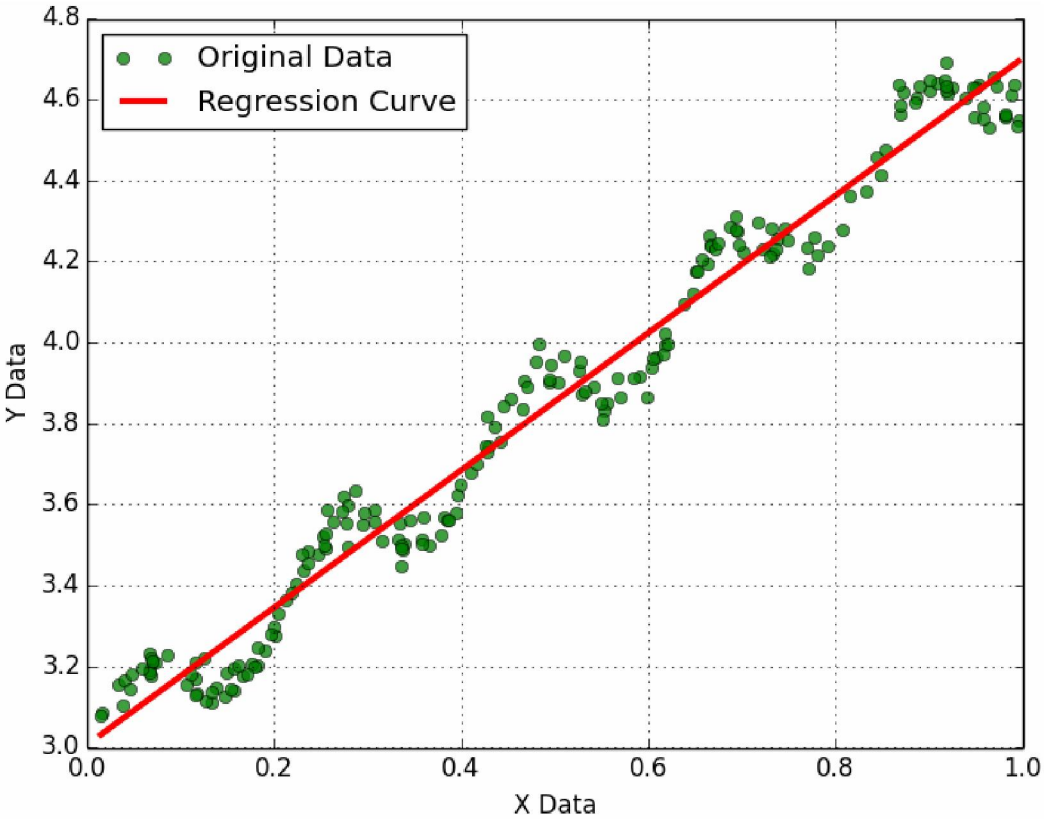
- 对于任意的(x,y) ∈ domf

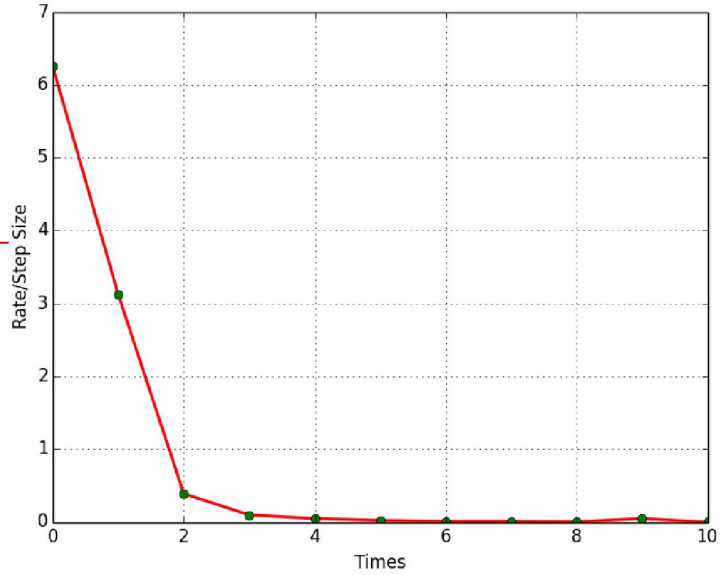
$$f(x,y) \le \max_x f(x,y)$$

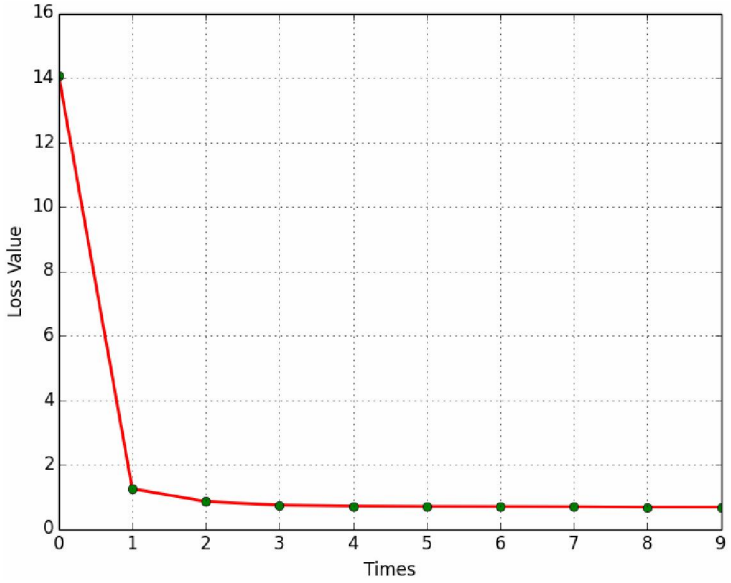
$$\Rightarrow \min_y f(x,y) \le \min_y \max_x f(x,y)$$

$$\Rightarrow \max_x \min_y f(x,y) \le \min_y \max_x f(x,y)$$

线性回归、rate、Loss





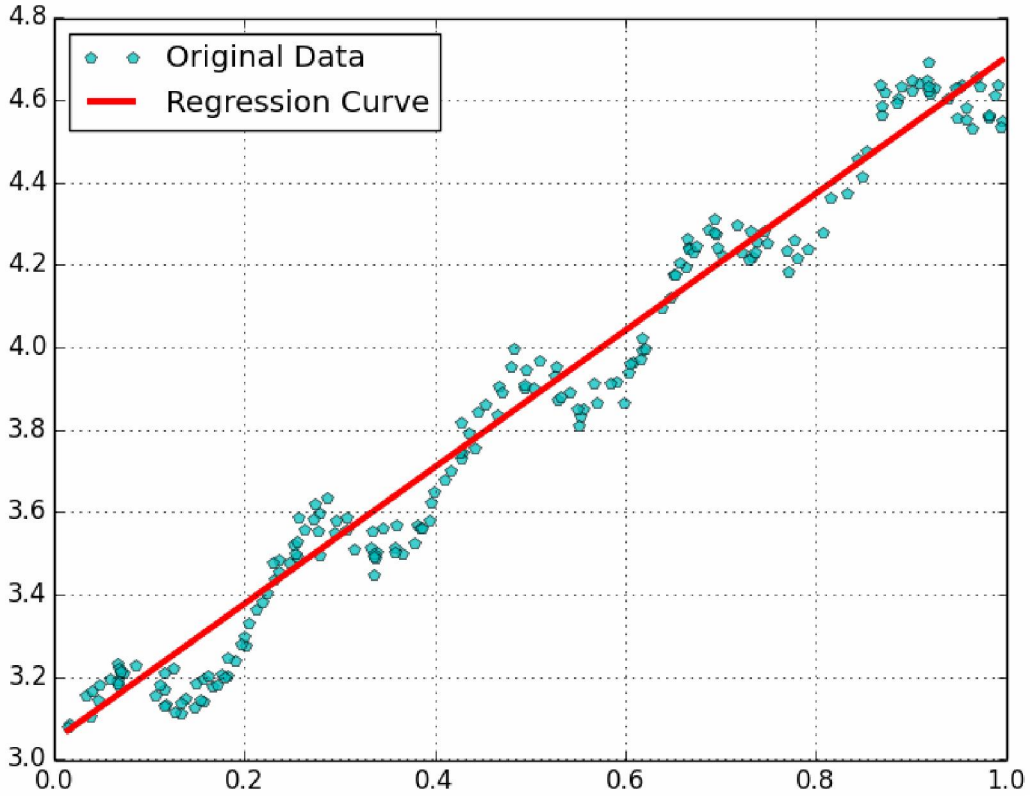


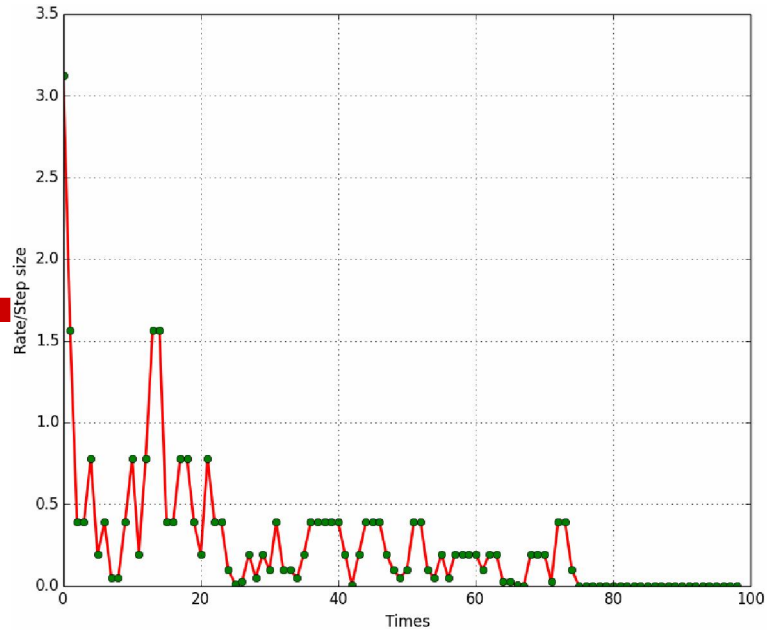
SGD与学习率

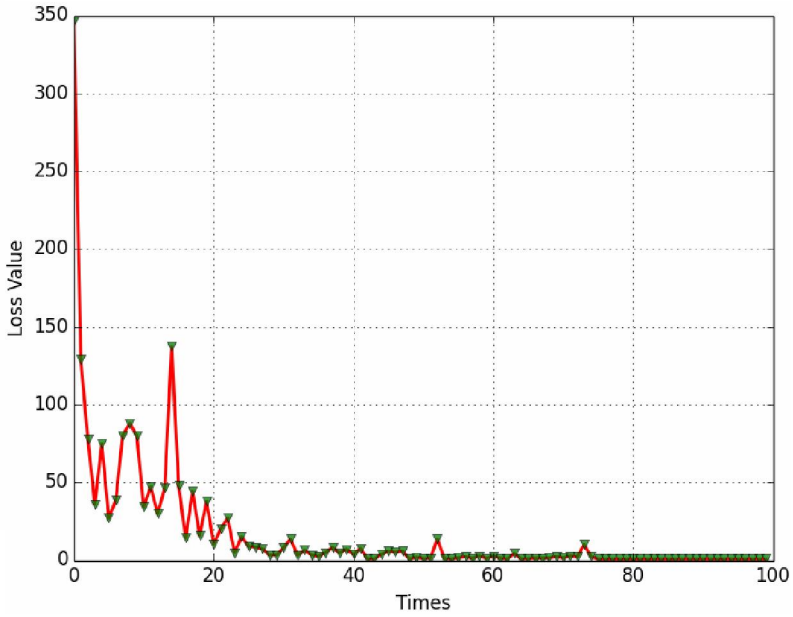
```
# w当前值；g当前梯度方向；a当前学习率；data数据
def calcAlphaStochastic(w, g, a, data):
    c1 = 0.01    # 因为是每个样本都下降，所以参数运行度大些，即：激进
    now = fwStochastic(w, data)
    wNext = assign(w)
    numberProduct(a, g, wNext)
    next = fwStochastic(wNext, data)
    # 寻找足够大的a，使得h(a)>0
    count = 30
    while next < now:
        if a < 1e-10:
            a = 0.01
        else:
            a *= 2
        wNext = assign(w)
        numberProduct(a, g, wNext)
        next = fwStochastic(wNext, data)
        count -= 1
        if count == 0:
            break

    # 寻找合适的学习率a
    count = 50
    while next > now - c1*a*dotProduct(g, g):
        a /= 2
        wNext = assign(w)
        numberProduct(a, g, wNext)
        next = fwStochastic(wNext, data)

        count -= 1
        if count == 0:
            break
    return a
```

```
def calcCoefficient(data, listA, listW, listLostFunction):
    M = len(data)       # 样本数目
    N = len(data[0])    # 维度
    w = [0 for i in range(N)]
    wNew = [0 for i in range(N)]
    g = [0 for i in range(N)]

    times = 0
    alpha = 100.0  # 学习率随意初始化
    same = False
    while times < 10000:
        i = 0
        while i < M:
            j = 0
            while j < N:
                g[j] = gradientStochastic(data[i], w, j)
                j += 1
            normalize(g)  # 正则化梯度
            alpha = calcAlphaStochastic(w, g, alpha, data[i])
            #alpha = 0.01
            numberProduct(alpha, g, wNew)

            print "times,i, alpha,fw,w,g:\t", times, i, alpha, fw(w, data), w, g
            if isSame(w, wNew):
                if times > 5:  #防止训练次数过少
                    same = True
                    break
            assign2(w, wNew)  # 更新权值

            listA.append(alpha)
            listW.append(assign(w))
            listLostFunction.append(fw(w, data))

            i += 1
        if same:
            break
        times += 1
    return w
```

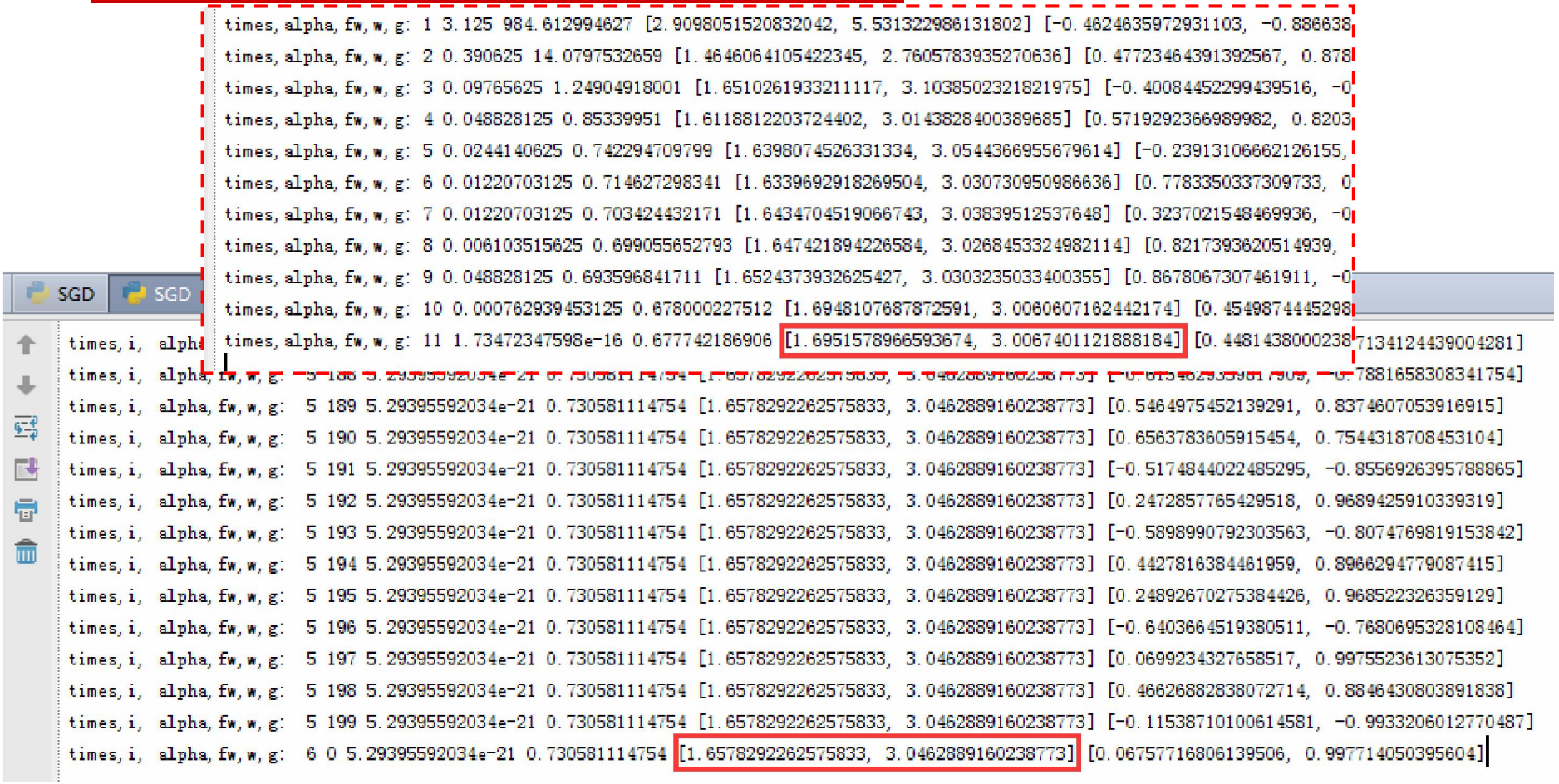
SGD


Original Data
Regression Curve
4.8
4.6
4.4
4.2
4.0
3.8
3.6
3.4
3.2
3.0
0.0
0.2
0.4
0.6
0.8
1.0


Rate/Step size
Times
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
20
40
60
80
100


Loss Value
Times
350
300
250
200
150
100
50
0
0
20
40
60
80
100

批处理与随机梯度下降


SGD
SGD
times, alpha, fw, w, g: 1 3.125 984.612994627 [2.9098051520832042, 5.531322986131802] [-0.4624635972931103, -0.886638
times, alpha, fw, w, g: 2 0.390625 14.0797532659 [1.4646064105422345, 2.7605783935270636] [0.47723464391392567, 0.878
times, alpha, fw, w, g: 3 0.09765625 1.24904918001 [1.6510261933211117, 3.1038502321821975] [-0.40084452299439516, -0
times, alpha, fw, w, g: 4 0.048828125 0.85339951 [1.6118812203724402, 3.0143828400389685] [0.5719292366989982, 0.8203
times, alpha, fw, w, g: 5 0.0244140625 0.742294709799 [1.6398074526331334, 3.0544366955679614] [-0.23913106662126155,
times, alpha, fw, w, g: 6 0.01220703125 0.714627298341 [1.6339692918269504, 3.030730950986636] [0.7783350337309733, 0
times, alpha, fw, w, g: 7 0.01220703125 0.703424432171 [1.6434704519066743, 3.03839512537648] [0.3237021548469936, -0
times, alpha, fw, w, g: 8 0.006103515625 0.699055652793 [1.647421894226584, 3.0268453324982114] [0.8217393620514939,
times, alpha, fw, w, g: 9 0.048828125 0.693596841711 [1.6524373932625427, 3.0303235033400355] [0.8678067307461911, -0
times, alpha, fw, w, g: 10 0.000762939453125 0.678000227512 [1.6948107687872591, 3.0060607162442174] [0.4549874445298
times, alpha, fw, w, g: 11 1.73472347598e-16 0.677742186906 [1.6951578966593674, 3.0067401121888184] [0.4481438000238713412443900428 1]
times, i, alpha, fw, w, g: 5 189 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [0.5464975452139291, 0.8374607053916915]
times, i, alpha, fw, w, g: 5 190 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [0.6563783605915454, 0.7544318708453104]
times, i, alpha, fw, w, g: 5 191 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [-0.5174844022485295, -0.8556926395788865]
times, i, alpha, fw, w, g: 5 192 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [0.2472857765429518, 0.9689425910339319]
times, i, alpha, fw, w, g: 5 193 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [-0.5898990792303563, -0.8074769819153842]
times, i, alpha, fw, w, g: 5 194 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [0.4427816384461959, 0.896629477908741 5]
times, i, alpha, fw, w, g: 5 195 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [0.24892670275384426, 0.968522326359129]
times, i, alpha, fw, w, g: 5 196 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [-0.6403664519380511, -0.7680695328108464]
times, i, alpha, fw, w, g: 5 197 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [0.0699234327658517, 0.9975523613075352]
times, i, alpha, fw, w, g: 5 198 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [0.46626882838072714, 0.8846430803891838]
times, i, alpha, fw, w, g: 5 199 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [-0.11538710100614581, -0.9933206012770487]
times, i, alpha, fw, w, g: 6 0 5.29395592034e-21 0.730581114754 [1.6578292262575833, 3.0462889160238773] [0.06757716806139506, 0.997714050395604]

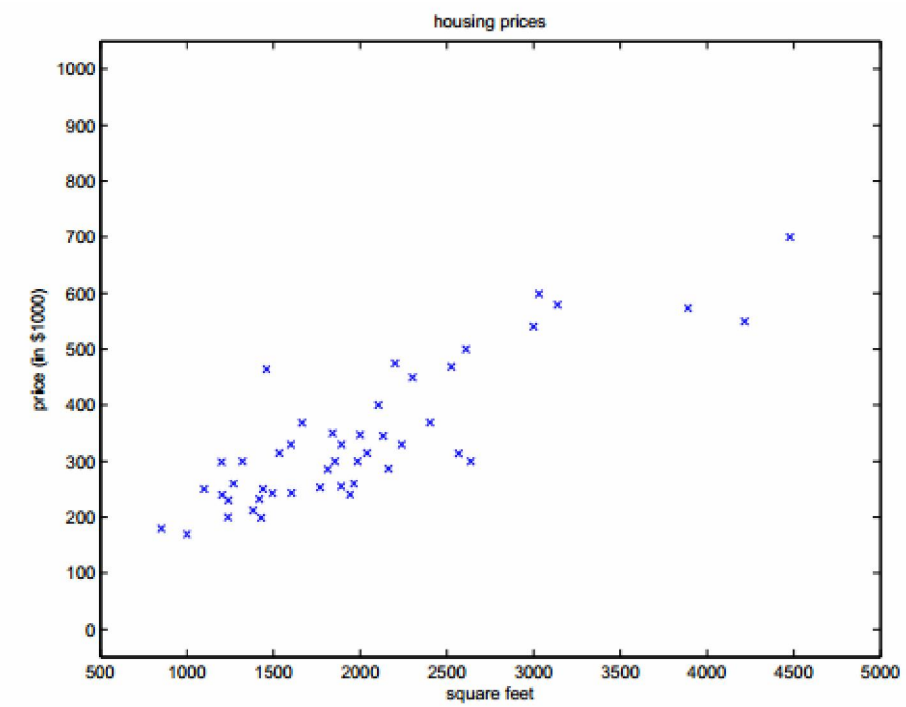
预备题目

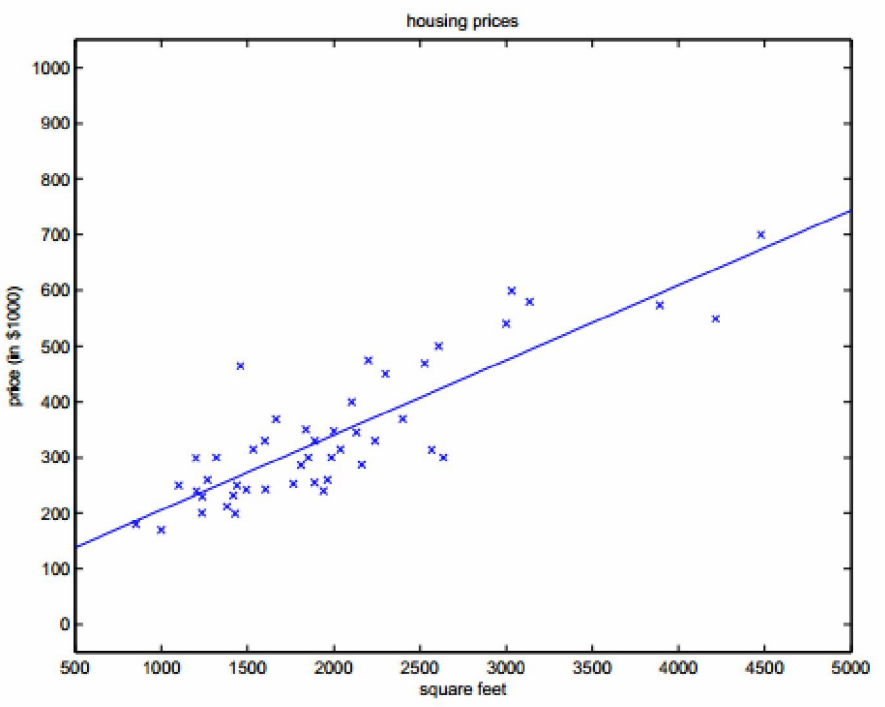
- 已知二次函数的一个点函数值和导数值，以及另外一个点的函数值，如果确定该函数的解析式？
  - 即：二次函数f(x)，已知f(a)，f'(a)，f(b)，求f(x)
  - 特殊的，若a=0，题目变成：
  - 对于二次函数f(x)，已知f(0)，f'(0)，f(a)，求f(x)

$$f(x)=\frac{f(a)-f'(0)a-f(0)}{a^2}x^2+f'(0)x+f(0)$$

从线性回归谈起

- $y=ax+b=ax_1+bx_0$，其中，$x_0 \equiv 1$

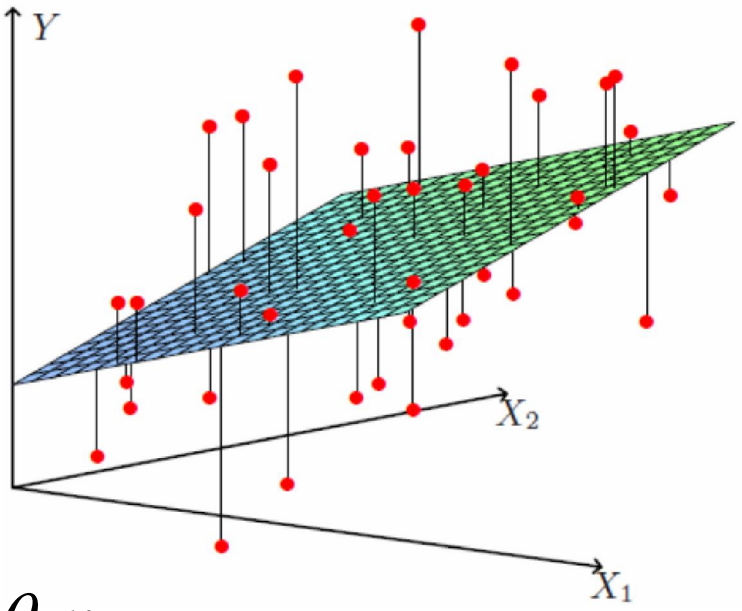




多个变量的情形

考虑两个变量

Living area (feet^2)	#bedrooms	Price (1000$s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
⋮	⋮	⋮



$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x$$

θ的解析式的求解过程

- 将M个N维样本组成矩阵X：
  - 其中，X的每一行对应一个样本，共M个样本
  - X的每一列对应样本的一个维度，共N维
    - 其实还有额外的一维常数项，全为1
- 一个比较“符合常理”的误差函数为：

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

- 已经论证：最小二乘建立的目标函数，是在高斯噪声的假设下，利用极大似然估计的方法建立的。

梯度下降算法 $J(\theta)=\frac{1}{2}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)^2$

- 初始化θ(随机初始化)
- 迭代，新的θ能够使得J(θ)更小
- 如果J(θ)无法继续减少或者达到循环上界次数，退出。

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

  - α：学习率、步长

线性回归目标函数的梯度方向计算

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

问题似乎完美解决

- 算法描述+凸函数极值

Repeat until convergence {

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

}

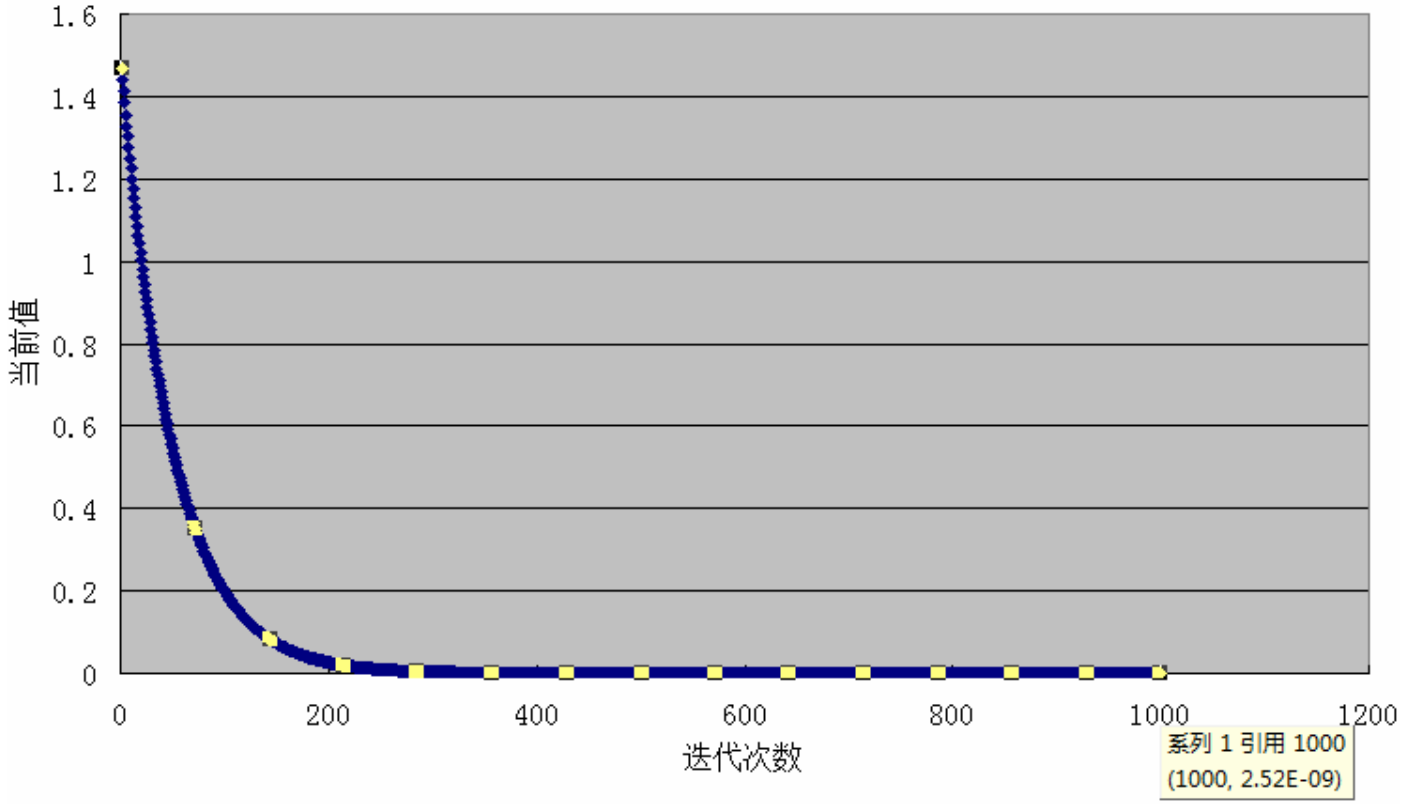
gradient descent. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function.

思考

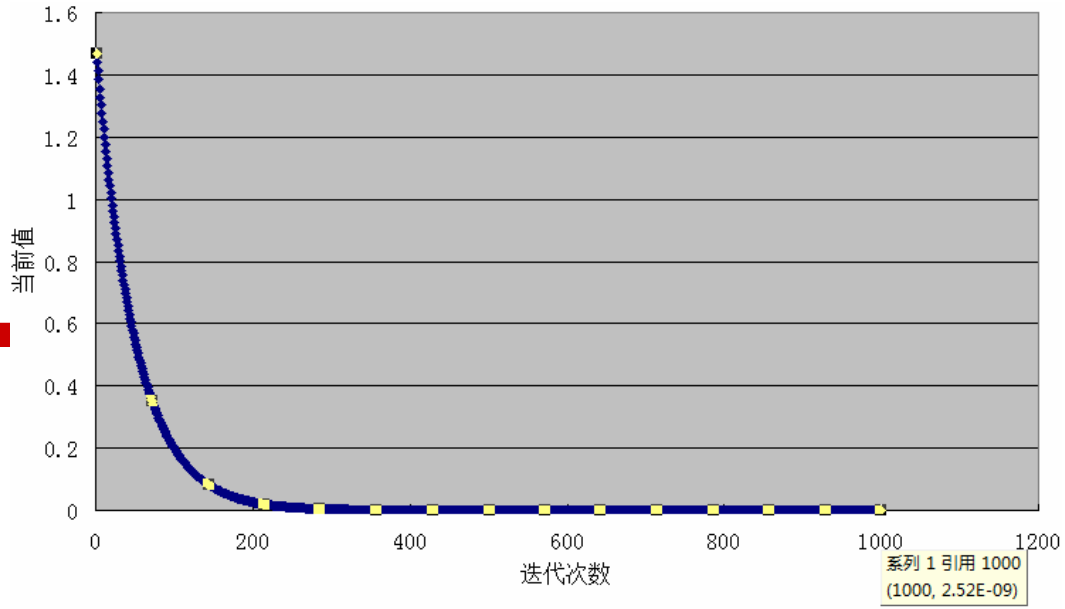
- 学习率α如何确定
  - 使用固定学习率还是变化学习率
  - 学习率设置多大比较好？
- 下降方向
  - 处理梯度方向，其他方向是否可以？
  - 可行方向和梯度方向有何关系？

实验：固定学习率的梯度下降

- $y=x^2$，初值取x=1.5，学习率使用0.01



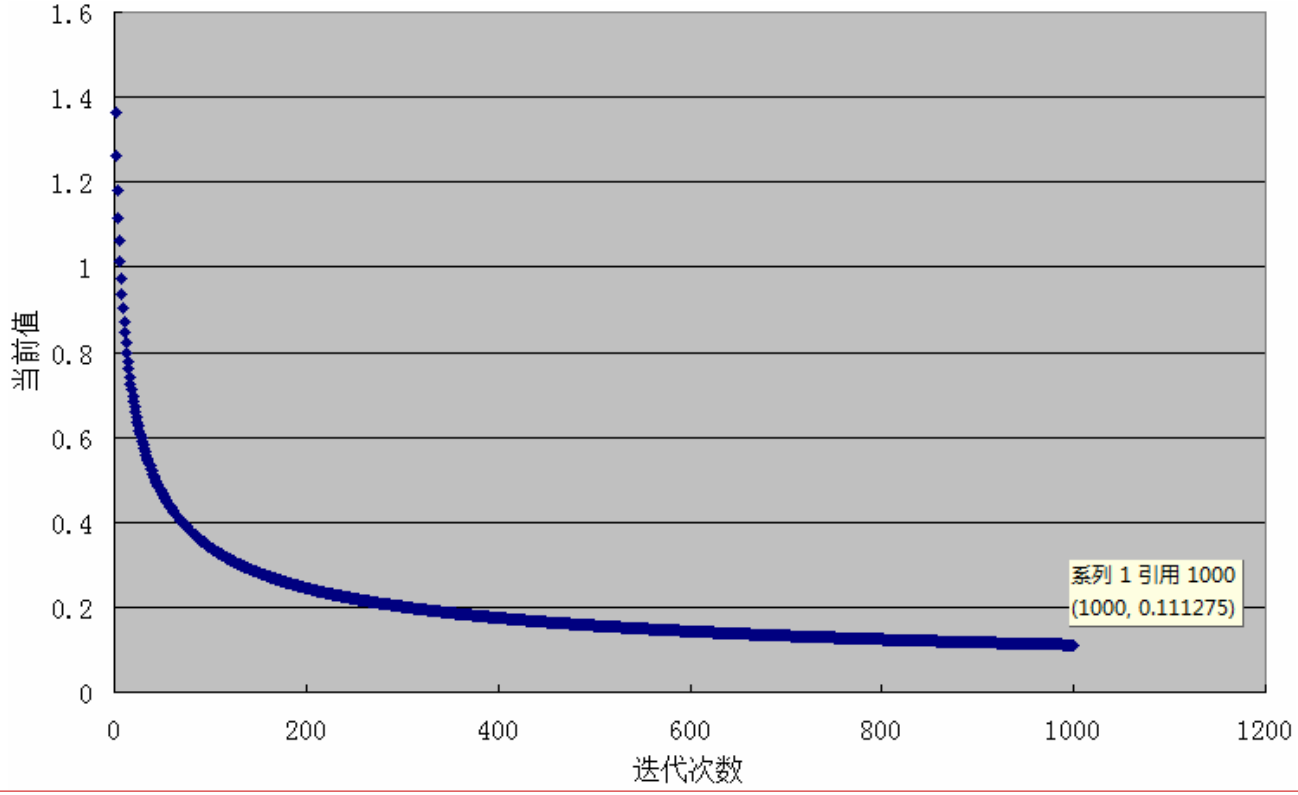
固定学习率



- 分析：
- 效果还不错
- 经过200次迭代，x=0.0258543；
- 经过1000次迭代，$x=2.52445\times10^{-9}$

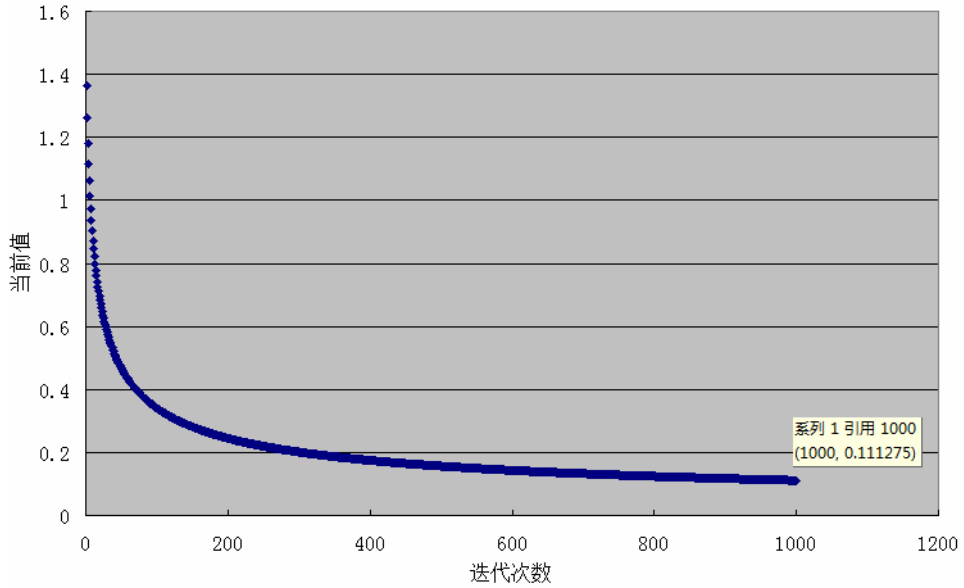
实验2：固定学习率的梯度下降

- $y=x^4$，初值取x=1.5，学习率使用0.01



固定学习率

- 分析：效果不理想
  - 经过200次迭代，x=0.24436；
  - 经过1000次迭代，x=0.111275



附：固定学习率实验的C代码

```
int _tmain(int argc, _TCHAR* argv[])
{
    double x = 1.5;
    double d;                  //一阶导
    double a = 0.01;           //学习率
    for(int i = 0; i < 1000; i++)
    {
        d = g(x);
        x -= d * a;
        cout << i << '\t' << a << '\t' << x << '\n';
    }
    return 0;
}
```

优化学习率

- 分析“学习率α”在f(x)中的意义
- 调整学习率：
  - 在斜率(方向导数)大的地方，使用小学习率
  - 在斜率(方向导数)小的地方，使用大学习率
- 如何构造学习率α

梯度下降的运行过程分析

- x_k=a，沿着负梯度方向，移动到x_{k+1}=b，有：

$$b = a - \alpha \nabla F(a) \Rightarrow f(a) \geq f(b)$$

- 从x0为出发点，每次沿着当前函数梯度反方向移动一定距离αk，得到序列：

$$x_0, x_1, x_2, \cdots, x_n$$

- 对应的各点函数值序列之间的关系为：

$$f(x_0) \geq f(x_1) \geq f(x_2) \geq \cdots \geq f(x_n)$$

- 当n达到一定值时，函数f(x)收敛到局部最小值

视角转换

- 记当前点为$\mathbf{x_k}$，当前搜索方向为$\mathbf{d_k}$(如：负梯度方向)，因为学习率α是待考察的对象，因此，将下列函数$f(\mathbf{x_k}+\alpha\,\mathbf{d_k})$看做是关于$\alpha$的函数$h(\alpha)$。

$$h(\alpha)=f(x_k+\alpha d_k),\quad \alpha>0$$

- 当$\alpha=0$时，$h(0)=f(\mathbf{x_k})$
- 导数 $\nabla h(\alpha)=\nabla f(x_k+\alpha d_k)^T d_k$

学习率α的计算标准

- 因为梯度下降是寻找f(x)的最小值，那么，在$\mathbf{x_k}$和$\mathbf{d_k}$给定的前提下，即寻找函数$f(\mathbf{x_k}+\alpha\,\mathbf{d_k})$的最小值。即：

$$\alpha = \arg\min_{\alpha>0} h(\alpha) = \arg\min_{\alpha>0} f(x_k + \alpha d_k)$$

- 进一步，如果h(α)可导，局部最小值处的α满足：

$$h'(\alpha) = \nabla f(x_k + \alpha d_k)^T d_k = 0$$

学习率函数导数的分析 $h'(\alpha)=\nabla f(x_k+\alpha d_k)^T d_k=0$

- 将 $\alpha=0$ 带入：

$$h'(0)=\nabla f(x_k+0*d_k)^T d_k=\nabla f(x_k)^T d_k$$

- 下降方向 $\mathbf{d_k}$ 可以选负梯度方向 $d_k=-\nabla f(x_k)$
  - 或者选择与负梯度夹角小于90°的某方向(后面会继续阐述搜索方向问题)
- 从而： $h'(0)<0$
- 如果能够找到足够大的 α，使得 $h'(\hat{\alpha})>0$
- 则必存在某 α，使得 $h'(\alpha^*)=0$
- α^* 即为要寻找的学习率。

线性搜索(Line Search)

- 最简单的处理方式
  - 二分线性搜索(Bisection Line Search)
  - 不断将区间[α1, α2]分成两半，选择端点异号的一侧，知道区间足够小或者找到当前最优学习率。

回溯线性搜索(Backing Line Search)

- 基于Armijo准则计算搜素方向上的最大步长，其基本思想是沿着搜索方向移动一个较大的步长估计值，然后以迭代形式不断缩减步长，直到该步长使得函数值$f(\mathbf{x_k}+\alpha\,\mathbf{d_k})$相对与当前函数值$f(\mathbf{x_k})$的减小程度大于预设的期望值(即满足Armijo准则)为止。

$$f(x_k+\alpha d_k)\le f(x_k)+c_1\alpha\nabla f(x_k)^T d_k \qquad c_1\in(0,1)$$

回溯与二分线性搜索的异同

- 二分线性搜索的目标是求得满足$h'(\alpha)\approx 0$的最优步长近似值，而回溯线性搜索放松了对步长的约束，只要步长能使函数值有足够大的变化即可。
- 二分线性搜索可以减少下降次数，但在计算最优步长上花费了不少代价；回溯线性搜索找到一个差不多的步长即可。

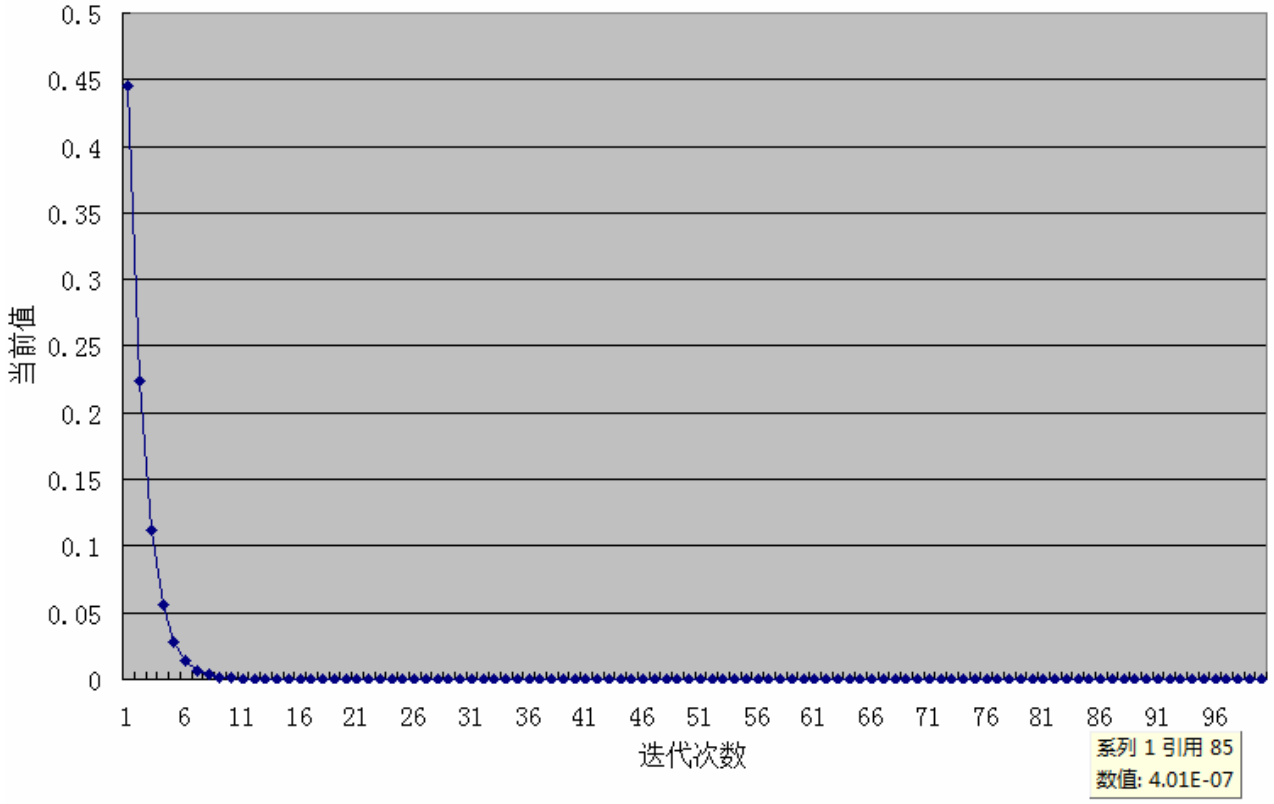
回溯线性搜索

- x为当前值
- d为x处的导数
- a为输入学习率
- 返回调整后的学习率

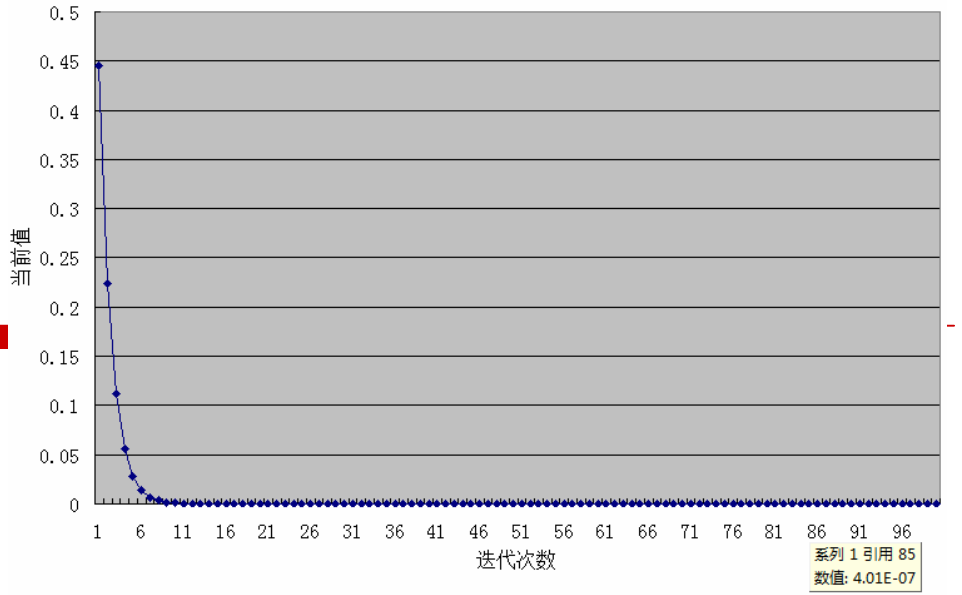
```
double GetA_Armijo(double x, double d, double a)
{
    double c1 = 0.3;
    double now = f(x);
    double next = f(x - a*d);

    int count = 30;
    while(next < now)
    {
        a *= 2;
        next = f(x - a*d);
        count--;
        if(count == 0)
            break;
    }

    count = 50;
    while(next > now - c1*a*d*d)
    {
        a /= 2;
        next = f(x - a*d);
        count--;
        if(count == 0)
            break;
    }
    return a;
}
```

实验：回溯线性搜索寻找学习率

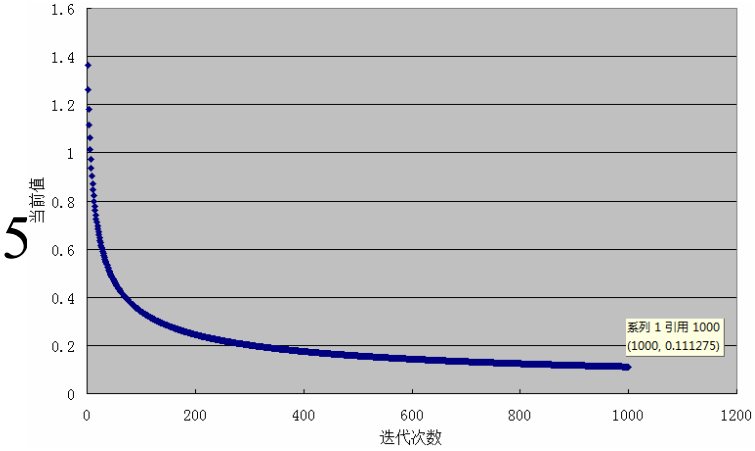
- $y=x^4$，初值取x=1.5，回溯线性方法



回溯线性搜索



- 分析：
- 效果还不错
- 经过12次迭代，x=0.00010872；
- 经过100次迭代，$x=3.64905\times10^{-7}$
  - 试比较固定学习率时：
    - 经过200次迭代，x=0.24436；
    - 经过1000次迭代，x=0.111275



回溯线性搜索的思考：插值法

- 采用多项式插值法(Interpolation)拟合简单函数，然后根据该简单函数估计函数的极值点，这样选择合适步长的效率会高很多。
- 现在拥有的数据为：x_k处的函数值$f(x_k)$及其导数$f'(x_k)$，再加上第一次尝试的步长α_0。如果α_0满足条件，显然算法退出；若α_0不满足条件，则根据上述信息可以构造一个二次近似函数：

$$h_q(\alpha)=\frac{h(\alpha_0)-h'(0)\alpha_0-h(0)}{\alpha_0^2}\alpha^2+h'(0)\alpha+h(0)$$

二次插值法求极值 $h_q(\alpha)=\frac{h(\alpha_0)-h'(0)\alpha_0-h(0)}{\alpha_0^2}\alpha^2+h'(0)\alpha+h(0)$

- 显然，导数为0的最优值为：

$$\alpha_1=\frac{h'(0)\alpha_0^2}{2[h'(0)\alpha_0+h(0)-h(\alpha_0)]}$$

- 若α_1满足Armijo准则，则输出该学习率；否则，继续迭代。

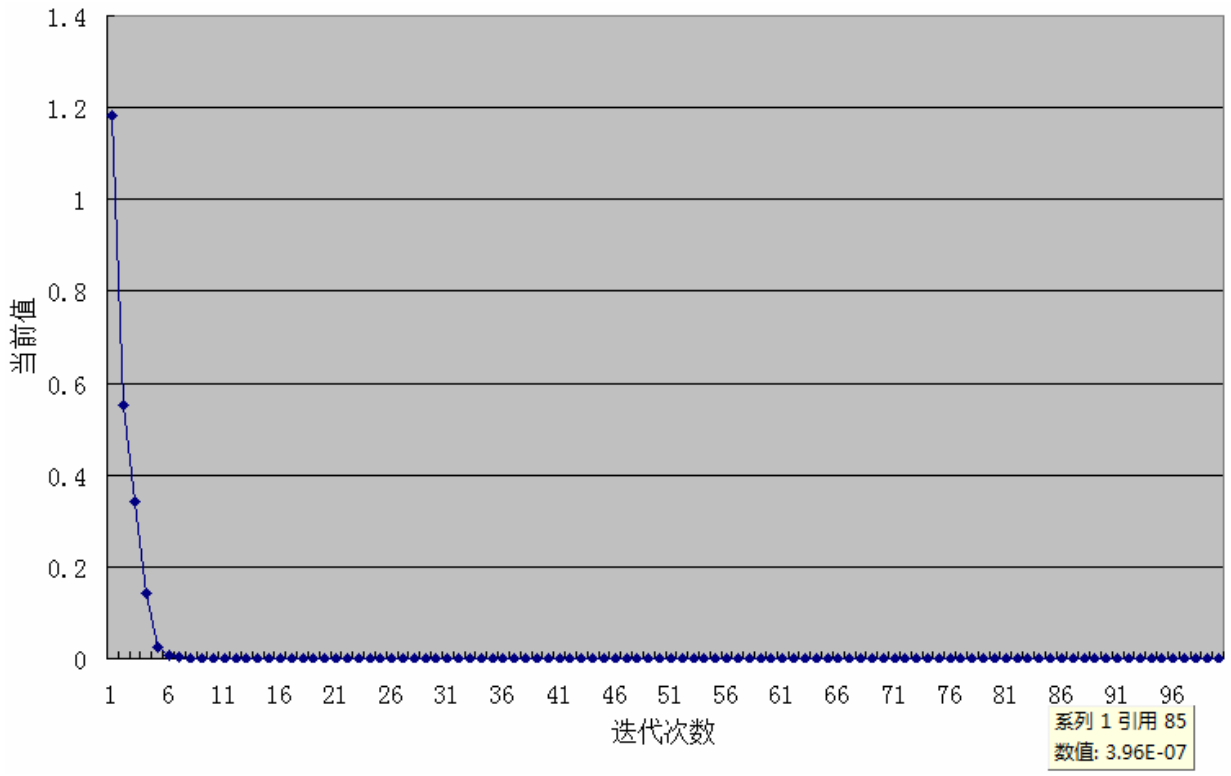
二次插值法

- x为当前值
- d为x处的导数
- a为输入学习率
- 返回调整后的学习率

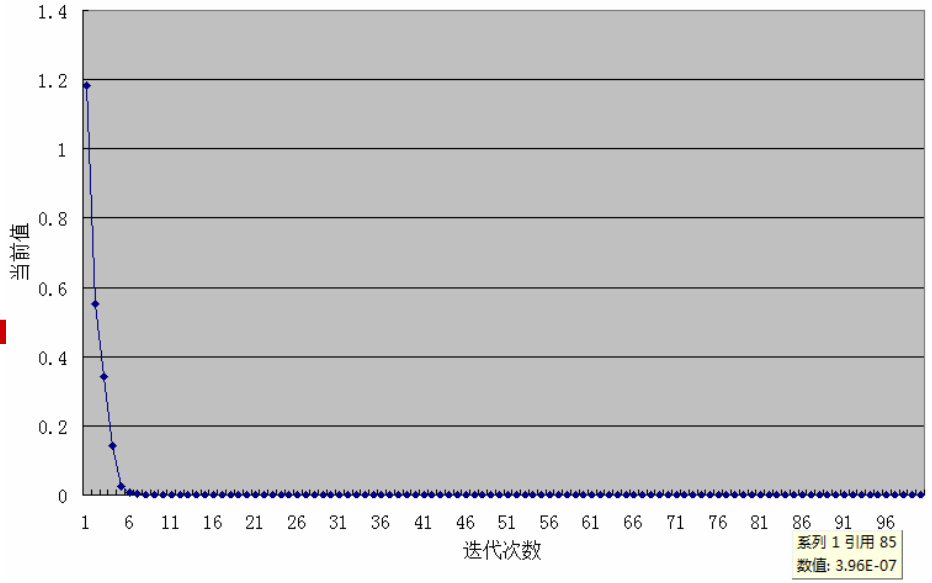
```
double GetA_Quad(double x, double d, double a)
{
    double c1 = 0.3;
    double now = f(x);
    double next = f(x - a*d);

    int count = 30;
    while(next < now)
    {
        a *= 2;
        next = f(x - a*d);
        count--;
        if(count == 0)
            break;
    }

    count = 50;
    double b;
    while(next > now - c1*a*d*d)
    {
        b = d * a * a / (now + d * a - next);
        b /= 2;
        if(b < 0)
            a /= 2;
        else
            a = b;
        next = f(x - a*d);
        count--;
        if(count == 0)
            break;
    }
    return a;
}
```

实验：二次插值线性搜索寻找学习率

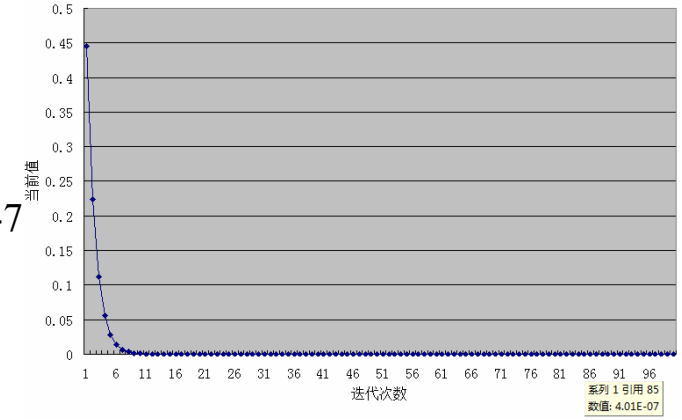
- $y=x^4$，初值取x=1.5，二次插值线性搜索方法



二次插值线性搜索



- 分析：
- 效果还不错
- 经过12次迭代，x=0.0000282229；
- 经过100次迭代，$x=3.61217\times10^{-7}$
  - 试比较回溯线性搜索时：
    - 经过12次迭代，x=0.00010872；
    - 经过1000次迭代，$x=3.649\times10^{-7}$



总结与思考

- 通过使用线性搜索的方式，能够比较好的解决学习率问题
- 一般的说，回溯线性搜索和二次插值线性搜索能够基本满足实践中的需要
- 问题：
  - 可否在搜索过程中，随着信息的增多，使用三次或者更高次的函数曲线，从而得到更快的学习率收敛速度？
  - 为避免高次产生的震荡，可否使用三次Hermite多项式，在端点保证函数值和一阶导都相等，从而构造更光顺的简单低次函数？

搜索方向

- 若搜索方向不是严格梯度方向，是否可以？
- 思考：
- 因为函数二阶导数反应了函数的凸凹性；二阶导越大，一阶导的变化越大。在搜索中，可否用二阶导做些“修正”？如：二者相除？

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

附：平方根算法

- 在任意点x_0处Taylor展开

令$f(x)=x^2-a$，即$f(x)=0$

$$\Rightarrow f(x)=f(x_0)+f'(x_0)(x-x_0)+o(x)$$

$$\Rightarrow f(x)\approx f(x_0)+f'(x_0)(x-x_0)$$

$$\Rightarrow 0=f(x_0)+f'(x_0)(x-x_0)$$

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

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

将$f(x_0)=x_0^2-a$和$f'(x_0)=2x_0$带入，

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

$$\Rightarrow x=x_0-\frac{x_0^2-a}{2x_0}$$

$$\Rightarrow x=\frac{1}{2}\left(x_0+\frac{a}{x_0}\right)$$

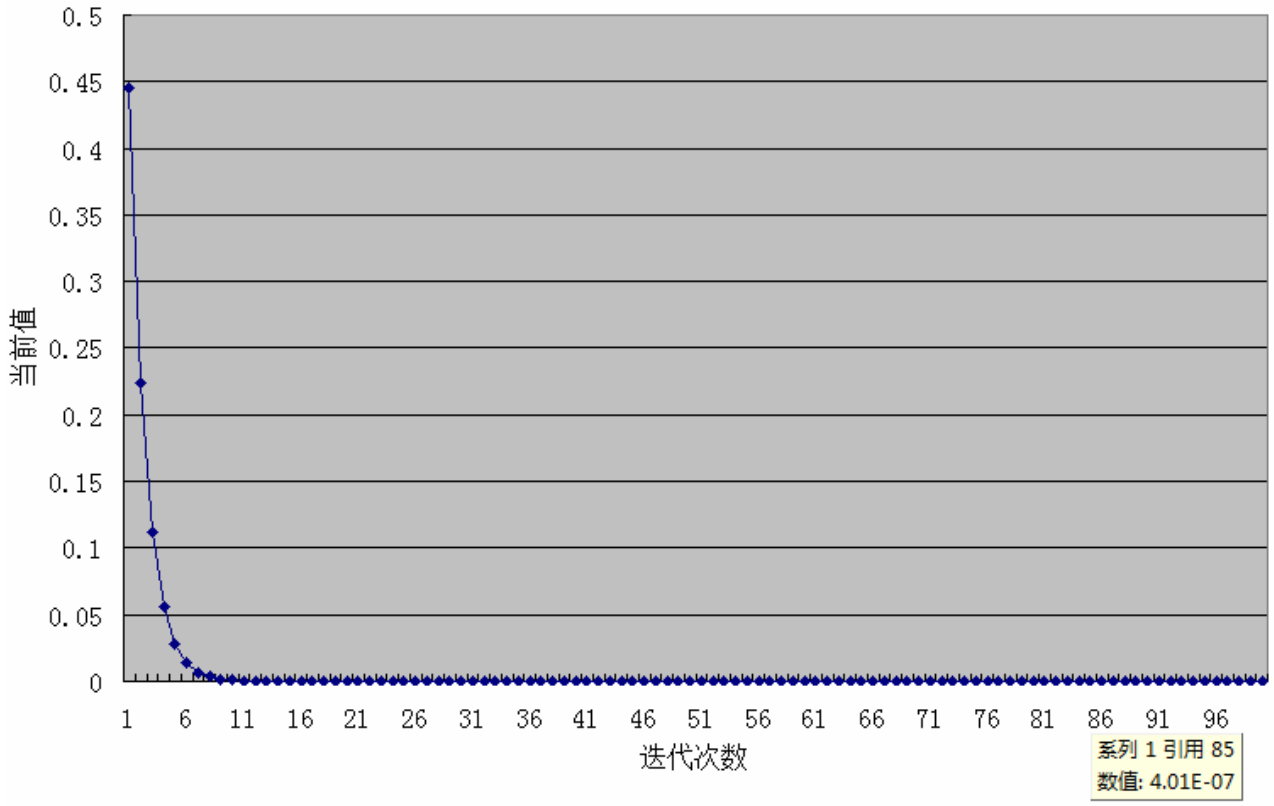
附：平方根

- 一般若干次(5、6次)迭代即可获得比较好的近似值。

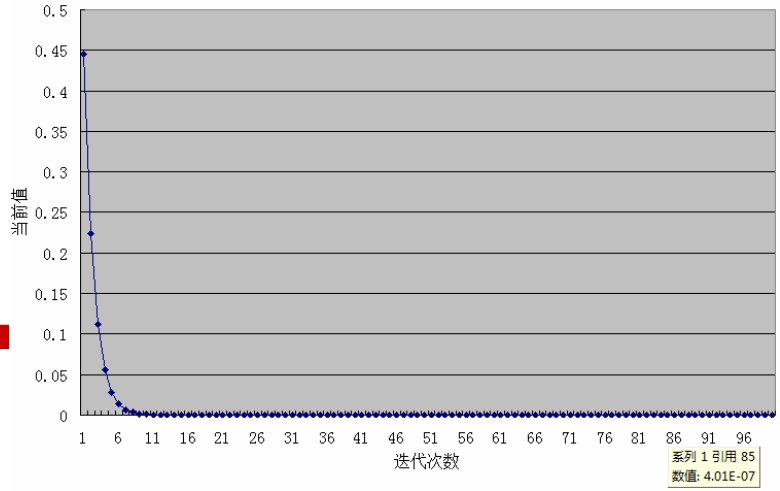
```
float Calc(float x);

int _tmain(int argc, _TCHAR* argv[])
{
    for(int i = 0; i <= 10; i++)
        cout << Calc((float)i) << '\n';
    return 0;
}

float Calc(float a)
{
    if(a < 1e-6)     //负数或者0，则直接返回0
        return 0;

    float x = a / 2;
    float t = a;
    while(fabs(x - t) > 1e-6)
    {
        t = x;
        x = (x + a/x) / 2;
    }
    return x;
}
```

实验：搜索方向的探索

- $y=x^4$，初值取x=1.5，负梯度除以二阶导



搜索方向的探索



- 效果出奇的好！
- 经过12次迭代，x=0.00770735；
- 经过100次迭代，$x=3.68948\times10^{-18}$
  - 试比较二次插值线性搜索时：
    - 经过12次迭代，x=0.0000282229；
    - 经过1000次迭代，$x=3.61217\times10^{-7}$

分析上述结果的原因

- 若f(x)二阶导连续，将f(x)在x_k处Taylor展开：

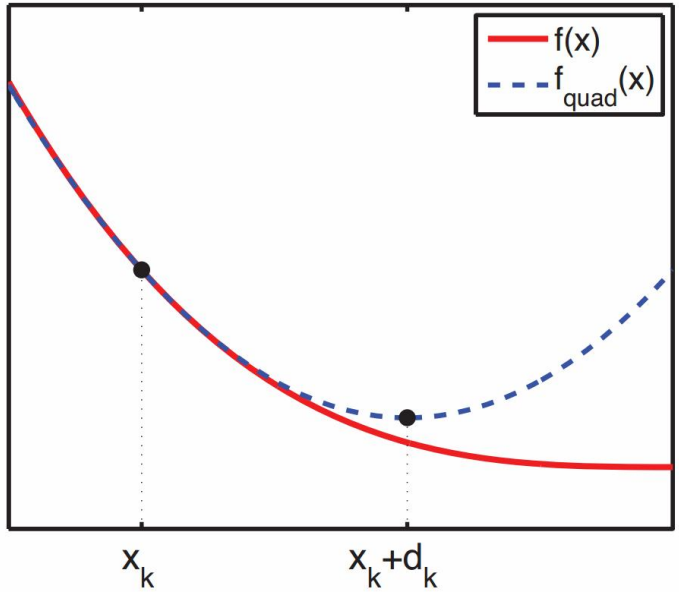
$$\varphi(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2} f''(x_k)(x - x_k)^2 + R_2(x)$$

$$\varphi'(x) \approx f'(x_k) + f''(x_k)(x - x_k)$$

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

牛顿法

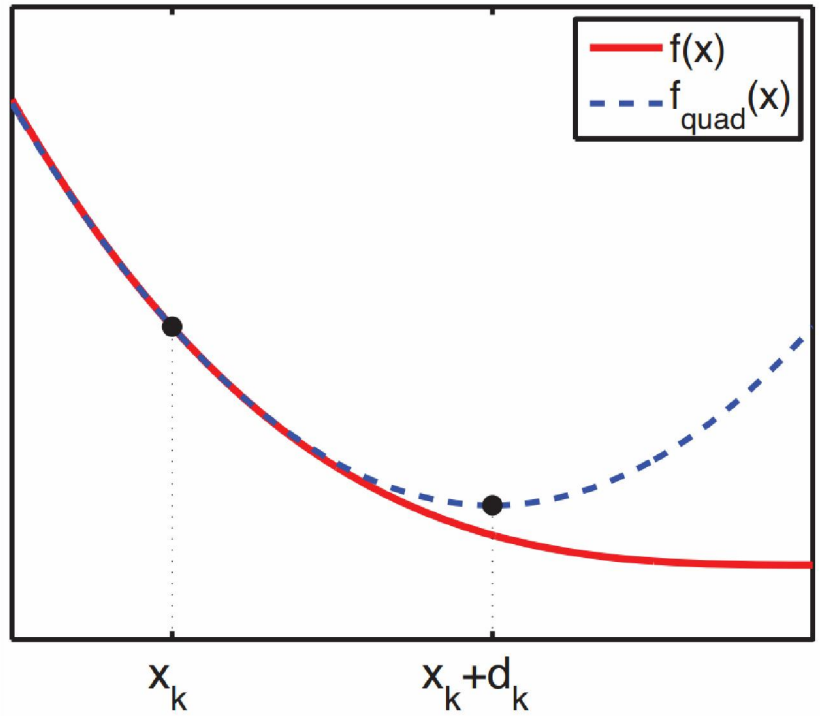
- 上述迭代公式，即牛顿法
- 该方法可以直接推广到多维：用方向导数代替一阶导，用Hessian矩阵代替二阶导

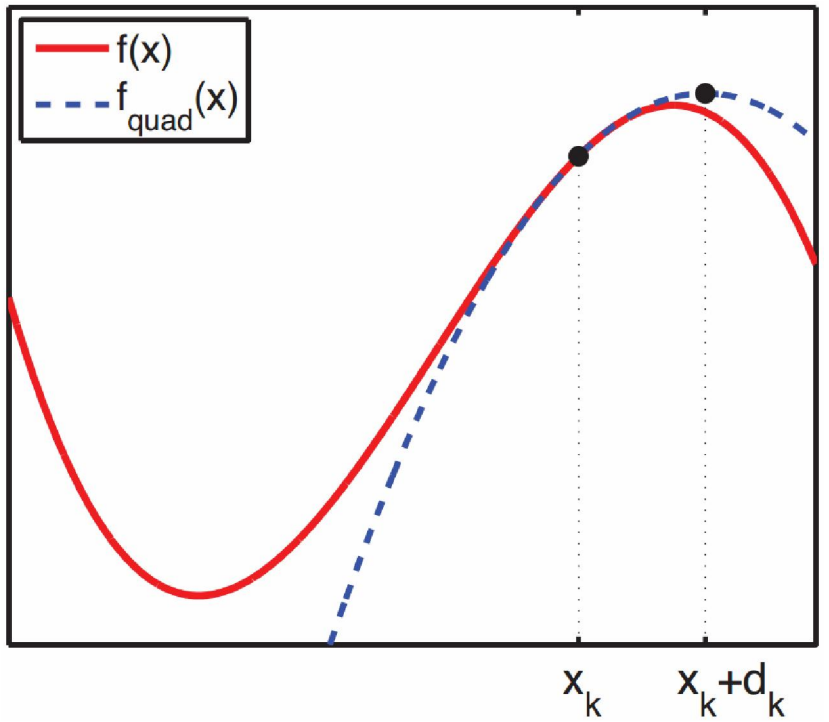


牛顿法的特点

- 经典牛顿法虽然具有二次收敛性，但是要求初始点需要尽量靠近极小点，否则有可能不收敛。
- 计算过程中需要计算目标函数的二阶偏导数，难度较大。
- 目标函数的Hessian矩阵无法保持正定，会导致算法产生的方向不能保证是f在x_k处的下降方向，从而令牛顿法失效；
- 如果Hessian矩阵奇异，牛顿方向可能根本是不存在的。

二阶导非正定的情况(一维则为负数)





修正牛顿方向

(1) Goldstein 和 Price 于 1967 年提出, 当 $\boldsymbol{G}_k$ 非正定时, 将搜索方向取为最速下降方向 $-\boldsymbol{g}^k$. 考虑到牛顿方向 $\boldsymbol{d}_k^N$ 与负梯度方向的夹角 $\langle \boldsymbol{d}_k^N, -\boldsymbol{g}^k \rangle$, 令搜索方向

$$\boldsymbol{d}^k = \begin{cases} \boldsymbol{d}_k^N, & \text{若 } \cos\langle \boldsymbol{d}_k^N, -\boldsymbol{g}^k \rangle \geqslant \eta > 0, \\ -\boldsymbol{g}^k, & \text{其他情形}, \end{cases}$$

其中 η 为某个事先设定的正数. 这样确定的搜索方向 $\boldsymbol{d}^k$ 满足 $\cos\langle \boldsymbol{d}^k, -\boldsymbol{g}^k \rangle \geqslant \eta > 0$, 从而可以保证算法的收敛性.

(2) Goldfeld 等人 (1966) 提出, 用正定矩阵 $\boldsymbol{G}_k + v_k \boldsymbol{I}$ 替代 Hesse 矩阵 $\boldsymbol{G}_k$, 计算修正牛顿方向. 比较理想的参数 $v_k > 0$ 满足: 适当大于使 $\boldsymbol{G}_k + v\boldsymbol{I}$ 正定的 "最小" 的 v. 此时, 可以借助于修正 Cholesky 分解算法确定参数 v_k.

拟牛顿的思路

- 求Hessian矩阵的逆影响算法效率，同时，搜索方向只要和负梯度的夹角小于90°即可，因此，可以用近似矩阵代替Hessian矩阵，只要满足该矩阵正定、容易求逆，或者可以通过若干步递推公式计算得到。
  - BFGS / LBFGS
  - Broyden – Fletcher – Goldfarb - Shanno

BFGS

- 矩阵迭代公式

$$\begin{aligned}\mathbf{B}_{k+1} &= \mathbf{B}_k + \frac{\mathbf{y}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k} - \frac{(\mathbf{B}_k\mathbf{s}_k)(\mathbf{B}_k\mathbf{s}_k)^T}{\mathbf{s}_k^T\mathbf{B}_k\mathbf{s}_k} \\ \mathbf{s}_k &= \boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1} \\ \mathbf{y}_k &= \mathbf{g}_k - \mathbf{g}_{k-1}\end{aligned}$$

  - 初值选单位阵 $\mathbf{B}_0 = \mathbf{I}$

- 记：$\mathbf{C}_k \approx \mathbf{H}_k^{-1}$

- 有：

$$\mathbf{C}_{k+1} = \left(\mathbf{I} - \frac{\mathbf{s}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}\right)\mathbf{C}_k\left(\mathbf{I} - \frac{\mathbf{y}_k\mathbf{s}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}\right) + \frac{\mathbf{s}_k\mathbf{s}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}$$

L-BFGS

- BFGS需要存储n×n的方阵C_k用于近似Hessian阵的逆矩阵；而L-BFGS仅需要存储最近m(m约为10，m=20足够)个$(\Delta x, \Delta(\nabla f(x)))$用于近似$C_k$即可。
- L-BFGS的空间复杂度O(mn)，若将m看做常数，则为线性，特别适用于变量非常多的优化问题。

我们在这里

- 七月算法 http://www.julyedu.com/
  - 视频/课程/社区
- 七月题库APP：Android/iOS
  - http://www.julyapp.com/
- 微博
  - @研究者July
  - @七月题库
  - @邹博_机器学习
- 微信公众号
  - julyedu

感谢大家！

恳请大家批评指正！